

Ladle¹

Jacob Butcher

Computer Science Division
Department of Electrical Engineering and Computer Science
University of California
Berkeley, California 94720

November 11, 1989

Report No. UCB/CSD 89/519

PIPER WORKING PAPER 89-4

ABSTRACT

Ladle is a language for specifying the structure of certain kinds of formal languages. The name stands for LAnguage Description Language.

A Ladle specification defines two structural aspects of language representation: lexical and syntactic. (A semantic specification will be added in a future release.) The syntax description encompasses the abstract syntax of the language, the internal tree representation of this abstract syntax, and how to parse and unparse such syntax trees.

The Ladle processor transforms a language specification into a set of tables that are used by the interactive language-based editor Pan I to map between text and abstract syntax trees, using either bottom-up parsing or structural elaboration. Access to the tables is provided by a client interface for Ladle.

The report first gives some background information and discusses the functionality of the Ladle processor at a fairly high level. The theoretical basis for Ladle is described. Subsequent sections specify Ladle's input format and semantics, its output data and format, and the client interface to Ladle's output tables.

¹Sponsored by the Defense Advanced Research Projects Agency (DoD), monitored by Space and Naval Warfare Systems Command under Contract N00039-88-C-0292, and by a gift from Apple Computer Corp.

Report Documentation Page			Form Approved OMB No. 0704-0188	
Public reporting burden for the collection of information is estimated to average 1 hour per response, including the time for reviewing instructions, searching existing data sources, gathering and maintaining the data needed, and completing and reviewing the collection of information. Send comments regarding this burden estimate or any other aspect of this collection of information, including suggestions for reducing this burden, to Washington Headquarters Services, Directorate for Information Operations and Reports, 1215 Jefferson Davis Highway, Suite 1204, Arlington VA 22202-4302. Respondents should be aware that notwithstanding any other provision of law, no person shall be subject to a penalty for failing to comply with a collection of information if it does not display a currently valid OMB control number.				
1. REPORT DATE 11 NOV 1989	2. REPORT TYPE		3. DATES COVERED 00-00-1989 to 00-00-1989	
4. TITLE AND SUBTITLE Ladle			5a. CONTRACT NUMBER	
			5b. GRANT NUMBER	
			5c. PROGRAM ELEMENT NUMBER	
6. AUTHOR(S)			5d. PROJECT NUMBER	
			5e. TASK NUMBER	
			5f. WORK UNIT NUMBER	
7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES) University of California at Berkeley, Department of Electrical Engineering and Computer Sciences, Berkeley, CA, 94720			8. PERFORMING ORGANIZATION REPORT NUMBER	
9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)			10. SPONSOR/MONITOR'S ACRONYM(S)	
			11. SPONSOR/MONITOR'S REPORT NUMBER(S)	
12. DISTRIBUTION/AVAILABILITY STATEMENT Approved for public release; distribution unlimited				
13. SUPPLEMENTARY NOTES				
14. ABSTRACT Ladle is a language for specifying the structure of certain kinds of formal languages. The name stands for LAnguage Description LanguagE. A Ladle specification defines two structural aspects of language representation: lexical and syntactic. (A semantic specification will be added in a future release.) The syntax description encompasses the abstract syntax of the language, the internal tree representation of this abstract syntax and how to parse and unparse such syntax trees. The Ladle processor transforms a language specification into a set of tables that are used by the interactive language-based editor Pan I to map between text and abstract syntax trees, using either bottom-up parsing or structural elaboration. Access to the tables is provided by a client interface for Ladle. The report first gives some background information and discusses the functionality of the Ladle processor at a fairly high level. The theoretical basis for Ladle is described. Subsequent sections specify Ladle's input format and semantics, its output data and format, and the client interface to Ladle's output tables.				
15. SUBJECT TERMS				
16. SECURITY CLASSIFICATION OF:			17. LIMITATION OF ABSTRACT Same as Report (SAR)	18. NUMBER OF PAGES 48
a. REPORT unclassified	b. ABSTRACT unclassified	c. THIS PAGE unclassified		

0 Introduction

Ladle is a language for specifying the structure of languages. The name stands for **L**anguage **D**escription **L**anguag**E**. A Ladle specification defines two structural aspects of language representation: lexical and syntactic.² The Ladle processor transforms a language specification into a set of tables which contain enough information to represent instances of the language as text, as sequences of lexical objects such as identifiers, integers, etc., or as syntax trees, and to convert between these representations. The tables also permit direct construction and manipulation of syntax trees representing language instances. Access to the tables is provided by a client interface for Ladle.

Section 1 gives some background information and discusses the functionality of the Ladle processor at a fairly high level. Section 2 contains the theoretical basis for Ladle. Section 3 specifies Ladle's input format and semantics. Section 4 details Ladle's output data and format. Section 5 describes the client interface to Ladle's output tables. There are also appendices which contain important notational conventions, examples, and some auxiliary information. In particular, Appendix A describes many of the mathematical notations and conventions used throughout the document. The casual reader may wish to skim Section 2 or skip Sections 4 and 5 entirely.

1 Overview

The core of a Ladle language specification is the description of the syntactic structure of the language. This description specifies the abstract syntax of the language, the internal tree representation of this abstract syntax, and how to parse and unparse such syntax trees. While these aspects are closely tied together, each has a certain amount of flexibility independent of the others. The remainder of this section elaborates each of these aspects of syntax.

1.1 Abstract Syntax

The abstract syntax of a language is a description of the complete syntactic structure of the language as it is understood by the language's users. For example, an abstract syntax for a programming language must contain a structure for each kind of statement and expression in the language, and must include all of the language's keywords, such as `BEGIN` and `END`. Note that this definition contrasts with some other usages of the term "abstract syntax", where keywords, parentheses, and other purely syntactic symbols are omitted.

In Ladle, the abstract syntax of a language is defined by an extended context free grammar for the language, called the *abstract* grammar. The abstract grammar need not be in any particular grammatical class such as LL or LALR. It may even be ambiguous. The abstract grammar will typically contain exactly one rule for each construct in the language, e.g. a declaration list, a

²A semantic specification will be added in a future release.

conditional statement, an addition expression. However, almost any correct grammar can be used if the precise derivation of an instance of the language is not of interest. Note that since no restrictions are placed on abstract grammars, it is unnecessary to transform a desired grammar into one that will work for Ladle.

In Ladle, the structure of an instance of a language is defined by an abstract grammar derivation that rewrites the start symbol as that instance. An *abstract syntax tree* (or just *syntax tree*) represents an abstract grammar derivation that rewrites a single non-terminal. A derivation may rewrite a non-terminal besides the initial symbol of the grammar or may result in a phrasal form that is not a terminal string. The *rhs* of such a derivation is not an instance of the language, but a syntax tree representing that derivation is still valid. Thus a syntax tree does not necessarily represent an instance of a language, but may represent a structured instance fragment.

1.2 Syntax Tree Internal Representation

A syntax tree represents the structure of a phrasal form of a language as an abstract derivation. The *immediate sub-tree* of a node in a syntax tree is the sub-tree consisting of that node and its children. Conceptually, the correspondence between a syntax tree and a derivation is that each immediate sub-tree corresponds to the application of one rewrite rule. Thus the internal nodes of a syntax tree represent abstract non-terminals and the leaves represent abstract terminals and non-terminals. An internal node can also be considered to represent the rewrite rule represented by that node's immediate sub-tree. (Some leaf nodes can be similarly considered to represent empty rules, and therefore ϵ as well.) Note that the root of a syntax tree represents the *lhs* of the derivation represented by the tree, and the frontier of the tree represents the *rhs* of the derivation.

The internal representation (IR) of a syntax tree may be more compact than an exact representation of the syntax tree. An important technique for reducing tree size is to have each internal tree node designate the rewrite rule represented by the node's immediate sub-tree rather than that rule's non-terminal *lhs*, since the *lhs* is easily computed from the rule. (Again, a leaf node may designate an empty rule.) There are then two methods that can be used to compact an IR tree. One makes the tree less broad by eliminating terminal leaf nodes and the other makes the tree less deep by eliminating rule nodes.

An IR tree need not represent every terminal in a rewrite rule's *rhs* explicitly. The tree node representing a rule may have child sub-trees representing derivations that rewrite the symbols in the *rhs* of the rule. The node *must* have a child node for each non-terminal symbol in the *rhs*. However, terminals may be represented implicitly, so long as the terminal in a particular position in a given rule is always represented the same way. In a given language, usually a terminal will always be represented explicitly or always be represented implicitly, but it is possible to specify the representation on a rule by rule basis. Figure 1 shows the difference between explicit and implicit terminal representation of the rule $\langle stmt \rightarrow IF\ expr\ THEN\ stmt \rangle$.

An IR tree also need not represent every rewrite rule explicitly. A rule can be represented in any of three ways. Consider the rule $\langle expr \rightarrow identifier \rangle$, the possible tree representations of which are shown in Figure 2. Tree (a) is the strict representation, while trees (b) and (c) are smaller,

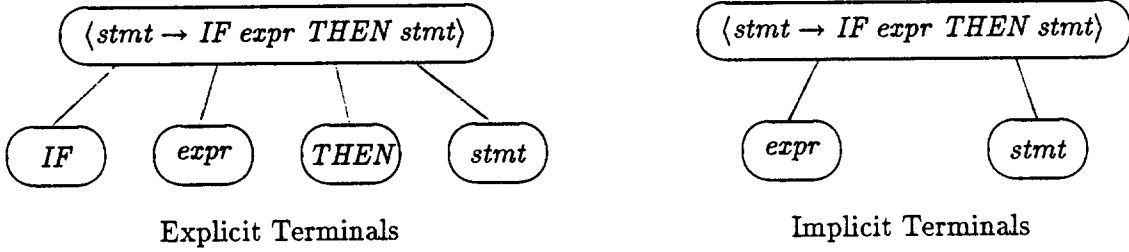


Figure 1: Two IR trees for the rule $\langle stmt \rightarrow IF\ expr\ THEN\ stmt \rangle$.

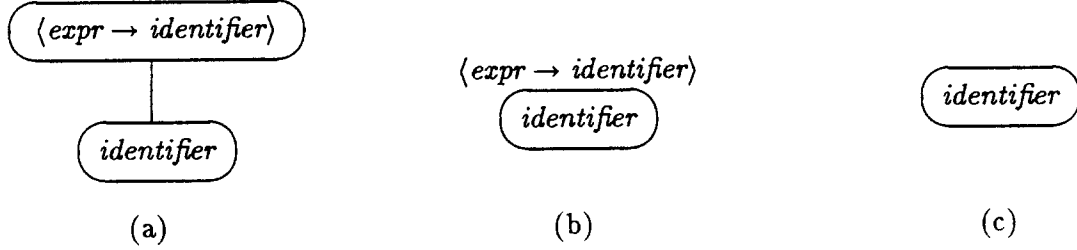


Figure 2: Three IR trees for the single rule derivation $\langle expr \rightarrow identifier \rangle$.

easier to use representations. Any rule \hat{p} whose rhs is not empty and contains no more than one non-terminal need not be represented by a tree node. Instead, \hat{p} may be represented by the node corresponding to the non-terminal in $rhs(\hat{p})$, or by the node corresponding to a specified explicitly represented terminal in the rhs if there is no non-terminal. As a representation of \hat{p} , this node may include the rule as an annotation or it may not, depending on the precise IR specified. In Figure 2, tree (b) includes the annotation, but tree (c) does not. The annotation must be included when $rhs(\hat{p})$ contains multiple symbols. Any number of annotations may be added to a node, in derivation order. Thus, a rule may be represented by a node, by an annotation on a node, or, if it is a chain rule, by nothing at all.

1.3 Parsing and Unparsing Syntax Trees

The abstract grammar used to define the syntax trees for a language should be clean and simple. However, there is no guarantee that it can be parsed easily, or even unambiguously. For this reason, Ladle language descriptions contain not just the abstract grammar, but a second context free grammar as well, called the *concrete* grammar.

The concrete grammar specifies how to convert an abstract phrasal form into an abstract syntax tree, and vice versa. The former operation is called *parsing*, and the latter *unparsing*. The concrete grammar must be LALR(1). The language specified by the concrete grammar must be the same one specified by the abstract grammar. In fact, the concrete grammar must be an *expansion* of the abstract grammar, a concept defined in Section 2. The parsing and unparsing operations are not specified explicitly, but are implicit in the relationship between the abstract and concrete grammars. Typically, the concrete grammar will be similar to the abstract one, but modified to include operator precedence, to have good error recovery properties, and to be LALR(1). However, the concrete

grammar can be any expansion of the abstract grammar, including the abstract grammar itself, if it is LALR(1).

2 Theory

The relationship between the abstract and concrete grammars in a Ladle language description specifies how to parse an abstract phrasal form into an abstract derivation, and also how to unparse an abstract derivation into an abstract phrasal form. This section defines the precise grammar relationship required, specifies the conversions, and provides algorithms for them. The approach is to parse with the concrete grammar, converting derivations and symbols between the grammars as necessary. (This section contains a great deal of notation. The reader may wish to review Appendix A before continuing.)

2.1 Grammatical Expansion

A context-free grammar \mathcal{G} may be said to be an *expansion* of another context-free grammar $\hat{\mathcal{G}}$ with respect to a mapping ψ_0 , which maps terminals and non-terminals of \mathcal{G} onto terminals and non-terminals of $\hat{\mathcal{G}}$, respectively, and a set \hat{P}_{cyclic} of $\hat{\mathcal{G}}$ rules, each of which has no semantic consequence. With respect to this expansion, the attributes (e.g. symbols, derivations, etc.) of $\hat{\mathcal{G}}$ are referred to as *abstract* attributes and the attributes of \mathcal{G} are *concrete*; that is, $\hat{\mathcal{G}}$ is the abstract grammar and \mathcal{G} is the concrete grammar. The conditions that define expansion ensure that the languages of $\hat{\mathcal{G}}$ and \mathcal{G} are identical, modulo the renaming of the mapping ψ_0 . These conditions further ensure that a concrete derivation can easily be transformed into an abstract derivation, and vice versa. Since the rules of \hat{P}_{cyclic} have no semantic consequence, these rules may be added or removed from an abstract derivation by such transformations.

Informally, to determine whether a concrete grammar \mathcal{G} is an expansion of an abstract grammar $\hat{\mathcal{G}}$ with respect to a mapping ψ_0 and a set of abstract rules \hat{P}_{cyclic} , perform the following steps:

- The domain of the mapping ψ_0 specifies a set of concrete non-terminals each of which is equivalent to an abstract non-terminal. Call this set \mathcal{A}_{base} .
- Extend \mathcal{A}_{base} to the set \mathcal{A}_{cyclic} by adding those concrete non-terminals that are cycle equivalent to non-terminals in \mathcal{A}_{base} . The cycle equivalence of two non-terminals is defined below; informally, each of the concrete non-terminals derives the other in a cycle using only chain rules and concrete derivations that correspond to abstract rules in \hat{P}_{cyclic} . It is for this reason that the set \hat{P}_{cyclic} must be specified as part of a grammatical expansion.
- Restrict \mathcal{A}_{cyclic} to the set \mathcal{A} by removing each concrete non-terminal whose only purpose is to form a cycle, that is, its only rewrite rule is part of the cycle that contains the non-terminal.
- Extend \mathcal{A} to the set \mathcal{A}_{chain} by adding concrete non-terminals that are chain equivalent to those in \mathcal{A} . The chain equivalence of two non-terminals is defined below, but loosely means that the non-terminal not in \mathcal{A} chain derives or is chain derived by the non-terminal that

is in \mathcal{A} . Each non-terminal in $\mathcal{A}_{cyclic} \setminus \mathcal{A}$ is chain-equivalent to some non-terminal in \mathcal{A} , so $\mathcal{A}_{cyclic} \subseteq \mathcal{A}_{chain}$.

- Construct Π , the set of concrete derivations each of which rewrites an abstracted concrete non-terminal as an abstracted phrasal form and which rewrites exactly one abstracted concrete non-terminal (the first). A phrasal form is abstracted when it contains only symbols in $(\Sigma \cup \mathcal{A}_{chain})$, and a concrete non-terminal is abstracted when it is in $(\Sigma \cup \mathcal{A}_{chain})$. Π is the set of concrete derivations that rewrite abstracted concrete phrasal forms as abstracted concrete sentential forms without rewriting any intermediate abstracted concrete phrasal forms.

\mathcal{G} is an expansion of $\hat{\mathcal{G}}$ when the concrete derivations of Π correspond precisely to the abstract rules of $\hat{\mathcal{G}}$, except for the concrete derivations that correspond to null abstract derivations.

DEFINITION: Grammatical Expansion

Let $\hat{\mathcal{G}} = \langle \hat{N}, \hat{\Sigma}, \hat{P}, \hat{S} \rangle$ and $\mathcal{G} = \langle N, \Sigma, P, S \rangle$ be the abstract and concrete context-free grammars, respectively.

Let $\mathcal{A}_{base} \subseteq N$ be a subset of concrete non-terminals.

Let $\psi_0 : (\Sigma \cup \mathcal{A}_{base}) \rightarrow (\hat{\Sigma} \cup \hat{N})$ be an invertible mapping such that $\psi_0(S) = \hat{S}$ and $\psi_0(Lang(\mathcal{G})) = Lang(\hat{\mathcal{G}})$.

$\mathcal{A}_{base} = \psi^{-1}(\hat{N})$ is the set of concrete non-terminals that correspond to abstract non-terminals.

The mapping ψ_0 allows the renaming of symbols between the two grammars.

Each abstract non-terminal derives a set of syntactic constructs for the specified language; each concrete non-terminal A in \mathcal{A}_{base} derives the concrete version of the set of constructs derived by the abstract non-terminal $\psi_0(A)$. (This statement is deliberately vague: the purpose of the remainder of this subsection is to couch the statement precisely.)

Let \hat{P}_{cyclic} be any set of abstract rules such that the only effect of applying a rule \hat{p} in \hat{P}_{cyclic} to an abstract phrasal form is to add some terminal symbols. These rules are *cyclic* and are chosen to describe syntactic constructs that may have no semantic importance, such as expression parentheses. More precisely, each rule \hat{p} in \hat{P}_{cyclic} must have the form $\hat{A} \rightarrow \hat{\xi}\hat{A}\hat{\zeta}$, where $\hat{A} \in \hat{N}$ and $\hat{\xi}, \hat{\zeta} \in \hat{\Sigma}^*$, although \hat{P}_{cyclic} need not contain all such rules. Removing an application of a cyclic abstract rule from an abstract derivation always yields another derivation.

Example:

$\hat{\mathcal{G}} = \langle \hat{N}, \hat{\Sigma}, \hat{P}, \hat{S} \rangle$, where $\hat{N} = \{\widehat{\text{stmt}}, \widehat{\text{expr}}\}$, $\hat{\Sigma} = \{\widehat{\text{name}}, \widehat{\text{IF}}, \widehat{\text{THEN}}, \widehat{\text{ELSE}}, \widehat{=}, \widehat{+}, \widehat{*}, \widehat{(}, \widehat{)}\}$, the rules of $\hat{P} = \{\hat{1}, \hat{2}, \hat{3}, \hat{4}, \hat{5}, \hat{6}, \hat{7}\}$ are

$\hat{1}$	$\widehat{\text{stmt}} \rightarrow \widehat{\text{name}} \widehat{=}$	$\widehat{\text{expr}}$
$\hat{2}$		$\widehat{\text{IF}} \widehat{\text{expr}} \widehat{\text{THEN}} \widehat{\text{stmt}}$
$\hat{3}$		$\widehat{\text{IF}} \widehat{\text{expr}} \widehat{\text{THEN}} \widehat{\text{stmt}} \widehat{\text{ELSE}} \widehat{\text{stmt}}$
$\hat{4}$	$\widehat{\text{expr}} \rightarrow \widehat{(}$	$\widehat{\text{expr}} \widehat{)}$
$\hat{5}$		$\widehat{\text{expr}} \widehat{+} \widehat{\text{expr}}$
$\hat{6}$		$\widehat{\text{expr}} \widehat{*} \widehat{\text{expr}}$
$\hat{7}$		$\widehat{\text{name}}$

and $\hat{S} = \widehat{\text{stmt}}$.

$\mathcal{G} = \langle N, \Sigma, P, S \rangle$, where $\Sigma = \{\text{name}, \text{IF}, \text{THEN}, \text{ELSE}, =, +, *, (,)\}$, $N = \{\text{stmt}, \text{assignment}, \text{ifstmt}, \text{elseclause}, \text{sup-expr}, \text{expr}, \text{term}, \text{factor}, \text{primary}\}$, the rules in $P = \{1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14\}$ are

1	stmt	→	assignment
2			ifstmt
3	assignment	→	name = sup-expr
4	ifstmt	→	IF sup-expr THEN stmt elseclause
5	elseclause	→	
6			ELSE stmt
7	sup-expr	→	expr
8	expr	→	expr + term
9			term
10	term	→	term * factor
11			factor
12	factor	→	primary
13	primary	→	name
14			(expr)

and $S = \text{stmt}$.

$\mathcal{A}_{\text{base}} = \{\text{stmt}, \text{expr}\}$.

$\forall X \in (\Sigma \cup \mathcal{A}_{\text{base}}), \psi_0(X) = \hat{X}$.

$\hat{P}_{\text{cyclic}} = \{\hat{4}\}$.

These grammars will be used as examples throughout the paper.

Two concrete non-terminals are *cycle equivalent* with respect to the abstract grammar when each derives the other using a cyclic concrete derivation, and some uniqueness constraints are satisfied. However, cyclic concrete derivations will not be defined until Section 2.2.1, since the definition depends on the set Π , which cannot be defined yet. Therefore, for the moment a concrete derivation $\pi \in P^*$ is said to be cyclic when there is an abstract derivation $\hat{\pi} \in \hat{P}_{\text{cyclic}}^*$ such that

$\psi_0(lhs(\pi)) \xrightarrow{\hat{\pi}} \psi_0(rhs(\pi))$. (P^* is defined in Appendix A.)

When two concrete non-terminals are cycle equivalent, both of them derive the same set of syntactic constructs, although there may be slight differences due to cyclic concrete derivations. Furthermore, they are characterized by a uniqueness condition on corresponding abstract derivations.

For each concrete non-terminal, *Cycled* defines the unique concrete non-terminal in \mathcal{A}_{base} which is cycle equivalent to it, if there is one.

Define *Cycled* : $N \rightarrow (\mathcal{A}_{base} \cup \{\perp\})$ by

$$\forall X \in N, \text{Cycled}(X) = \begin{cases} X & \text{if } X \in \mathcal{A}_{base} \\ A & \text{if } \exists! A \in \mathcal{A}_{base}, (A \Rightarrow \alpha \xrightarrow{P_{\neg \mathcal{A}_{base}}} \xi_0 X \zeta_0 \xrightarrow{P_{\neg \mathcal{A}_{base}}} \xi_0 \xi_1 A \zeta_1 \zeta_0, \text{ where} \\ & \xi_0, \xi_1, \zeta_0, \zeta_1 \in \Sigma^* \text{ and } \psi_0(A) \xrightarrow{\hat{P}_{cyclic}} \psi_0(\xi_0 \xi_1 A \zeta_1 \zeta_0)) \\ \perp & \text{otherwise.} \end{cases}$$

\mathcal{A}_{cyclic} is the set of concrete non-terminals which are cycle equivalent to a non-terminal in \mathcal{A}_{base} .

Note that it is not possible for two non-terminals in \mathcal{A}_{base} to be cycle equivalent to each other.

Define $\mathcal{A}_{cyclic} \subseteq N$ by $\mathcal{A}_{cyclic} = \text{Cycled}^{-1}(\mathcal{A}_{base})$.

Example:

$\text{Cycled}(\{\text{stmt}\}) = \{\text{stmt}\}$.

$\text{Cycled}(\{\text{expr}, \text{term}, \text{factor}, \text{primary}\}) = \{\text{expr}\}$.

$\text{Cycled}(\{\text{assignment}, \text{ifstmt}, \text{elseclause}, \text{sup-expr}\}) = \{\perp\}$.

$\mathcal{A}_{cyclic} = \{\text{stmt}, \text{expr}, \text{term}, \text{factor}, \text{primary}\}$.

Note that $\langle \text{expr} \xrightarrow{*} \text{factor} + \text{factor} \rangle$, while $\langle \text{term} \xrightarrow{*} (\text{factor} + \text{factor}) \rangle$. The derivations differ syntactically only by the presence of the parenthesis in the latter, which results from the cyclic concrete derivation corresponding to the abstract rule 4.

Two concrete non-terminals X and A can be *chain equivalent* to each other in two different ways. First of all, when A is the only concrete phrasal form that can be derived from X without rewriting a non-terminal in \mathcal{A}_{cyclic} after the first rewrite, X and A are chain equivalent. Secondly, for a given X , if no non-terminal satisfies the first condition and A is the only non-terminal which can derive a phrasal form containing X without rewriting a non-terminal in \mathcal{A}_{cyclic} after the first rewrite *and* the only phrasal form so derived is X , then X and A are also chain equivalent.

If X and A are chain equivalent, every terminal string derived from X is also derived from A .

\mathcal{A} is the set of concrete non-terminals each of which is cycle equivalent to a non-terminal in \mathcal{A}_{base} but is not chain equivalent to any non-terminal in \mathcal{A}_{cyclic} other than itself.

Define $\mathcal{A} \subseteq N$ by

$$\mathcal{A} = \left\{ A \in \mathcal{A}_{cyclic} \mid \neg \left((\exists! p \in P \text{ such that } lhs(p) = A) \text{ and } A \Rightarrow \gamma \xrightarrow{P_{\neg \mathcal{A}_{cyclic}}} B, \text{ where } B \in \mathcal{A}_{cyclic} \text{ and } \text{Cycled}(B) = \text{Cycled}(A) \right) \right\}.$$

\mathcal{A}_{base} is not necessarily a subset of \mathcal{A} . In fact, S itself may not be in \mathcal{A} .

Example:

$\mathcal{A} = \{\text{stmt}, \text{expr}, \text{term}, \text{primary}\}$.

The concrete non-terminal factor is not in \mathcal{A} because it is chain equivalent to primary.

To illustrate why \mathcal{A}_{base} may not be a subset of \mathcal{A} , consider the following modification of the example: Suppose that \mathcal{A}_{base} was defined as $\{\text{stmt}, \text{factor}\}$ instead of $\{\text{stmt}, \text{expr}\}$.

Both \mathcal{A}_{cyclic} and \mathcal{A} would still be the same. (In fact, *everything* that has been or will be defined would still be the same, except for \mathcal{A}_{base} .) However, now $\mathcal{A}_{base} \not\subseteq \mathcal{A}$.

Chained and \mathcal{A}_{chain} define the set of concrete non-terminals that are chain equivalent to the non-terminals of \mathcal{A} .

Define *Chained* : $N \rightarrow (\mathcal{A} \cup \{\perp\})$ by

$\forall X \in N,$

if $X \in \mathcal{A}$

Chained(X) = X

else if $(\exists! \alpha \in (\Sigma \cup \mathcal{A})^*, X \xrightarrow{P_{\mathcal{A}}} \alpha)$ and $\alpha = A$

Chained(X) = A

else if $(\exists! A \in \mathcal{A}, A \Rightarrow \alpha \xrightarrow{*} \xi X \zeta \text{ and } \xi, \zeta \in (\Sigma \cup \mathcal{A})^* \text{ and } \xi = \zeta = \epsilon)$

Chained(X) = A

otherwise

Chained(X) = \perp .

The use of “else” and “ $\exists!$ ” in this definition, rather than “or” and “ \exists ”, ensures that a concrete non-terminal is chain equivalent to at most one non-terminal of \mathcal{A} .

Define $\mathcal{A}_{chain} \subseteq N$ by $\mathcal{A}_{chain} = \text{Chained}^{-1}(\mathcal{A})$.

Example:

By the first condition,

Chained(stmt) = stmt

Chained(expr) = expr

Chained(term) = term

Chained(primary) = primary.

By the second condition,

Chained(sup-expr) = expr

Chained(factor) = primary.

By the third condition,

Chained(assignment) = stmt

Chained(ifstmt) = stmt.

By the final condition,

Chained(elseclause) = \perp .

$\mathcal{A}_{chain} = \{\text{stmt}, \text{assignment}, \text{ifstmt}, \text{sup-expr}, \text{expr}, \text{term}, \text{factor}, \text{primary}\}$.

Note that $\mathcal{A} \subseteq \mathcal{A}_{cyclic} \subseteq \mathcal{A}_{chain} \subseteq N$, since $\mathcal{A}_{cyclic} \setminus \mathcal{A}$ contains only concrete non-terminals that are chain equivalent to non-terminals of \mathcal{A} , and therefore $\mathcal{A}_{cyclic} \setminus \mathcal{A} \subset \mathcal{A}_{chain}$.

The mapping ψ extends the renaming mapping ψ_0 using the concrete non-terminal equivalences just defined.

Define $\psi : (\Sigma \cup \mathcal{A}_{chain}) \rightarrow (\hat{\Sigma} \cup \hat{N})$ by

$$\forall X \in (\Sigma \cup \mathcal{A}_{chain}), \quad \psi(X) = \begin{cases} \psi_0(X) & \text{if } X \in (\Sigma \cup \mathcal{A}_{base}) \\ \psi_0(\text{Cycled}(X)) & \text{if } X \in (\mathcal{A} \setminus \mathcal{A}_{base}) \\ \psi_0(\text{Cycled}(\text{Chained}(X))) & \text{if } X \in (\mathcal{A}_{chain} \setminus (\mathcal{A} \cup \mathcal{A}_{base})). \end{cases}$$

Note that $\psi \mathcal{A}_{chain}(\hat{\Sigma} \cup \hat{N}) = \psi_0 \circ \text{Cycled} \circ \text{Chained}$.

The concrete symbols in the set $(\Sigma \cup \mathcal{A}_{chain})$ are said to be *abstracted*, as each of them is mapped onto an abstract symbol. An abstracted concrete phrasal form is one that contains only abstracted concrete symbols.

Example:

$$\mathcal{A}_{base} = \{\text{stmt}, \text{expr}\}.$$

$$\psi(\text{stmt}) = \widehat{\text{stmt}}.$$

$$\psi(\text{expr}) = \widehat{\text{expr}}.$$

$$\mathcal{A} \setminus \mathcal{A}_{base} = \{\text{term}, \text{primary}\}.$$

$$\psi(\text{term}) = \widehat{\text{expr}}.$$

$$\psi(\text{primary}) = \widehat{\text{expr}}.$$

$$\mathcal{A}_{chain} \setminus (\mathcal{A} \cup \mathcal{A}_{base}) = \{\text{assignment}, \text{ifstmt}, \text{sup-expr}, \text{factor}\}.$$

$$\psi(\text{assignment}) = \widehat{\text{stmt}}.$$

$$\psi(\text{ifstmt}) = \widehat{\text{stmt}}.$$

$$\psi(\text{sup-expr}) = \widehat{\text{expr}}.$$

$$\psi(\text{factor}) = \widehat{\text{expr}}.$$

$N \setminus \mathcal{A}_{chain} = \{\text{elseclause}\}$, so ψ is not defined on the concrete non-terminal elseclause.

Π is the set of rightmost concrete derivations that rewrite an abstracted non-terminal as an abstracted phrasal form without rewriting any other abstracted non-terminals after the first.

Π_{base} is Π without certain chain derivations.

Define

$$\Pi_{base} = \left\{ \pi \in PP_{\neg \mathcal{A}_{chain}}^* \mid \begin{array}{l} \langle A \xRightarrow{*} A' \xRightarrow{\pi} \alpha' \xRightarrow{*} \alpha \rangle \in PP_{\neg \mathcal{A}}^*, \text{ where } A \in \mathcal{A}, A' \in \mathcal{A}_{chain}, \\ \alpha' \in (\Sigma \cup \mathcal{A}_{chain})^*, \alpha \in (\Sigma \cup \mathcal{A})^*, \psi(A') = \psi(A), \psi(\alpha') = \psi(\alpha), \\ \text{and } \pi \text{ is rightmost} \end{array} \right\}.$$

Define $\Pi = \Pi_{base} \cup \{\pi \in PP_{\neg \mathcal{A}}^* \mid A \xRightarrow{\pi} B \text{ where } A, B \in \mathcal{A}_{chain}, \psi(A) = \psi(B), \text{ and } \pi \text{ is rightmost}\}$.

Π and Π_{base} contain only rightmost derivations so that no two derivations in the same set perform the same rewrite in different orders.

Π_{base} , and hence Π , can be infinite in size. However, the next requirement imposed by grammatical abstraction will ensure that the derivations of Π_{base} and Π have a finite number of distinct *lhs* and *rhs*. Further, Π_{base} and Π can be infinite in size only if \mathcal{G} is ambiguous.

Example:

$\Pi_{base} = \{3, 4:5, 4:6, 8, 9, 10, 11, 13, 14\}$.

(The derivation consisting of rule 4 followed by rule 5 is represented by “4:5”, since “45” is potentially ambiguous.)

Else-clause is the only concrete non-terminal that is not abstracted, so only rules containing else-clause in their *rhs*'s result in Π_{base} derivations of length greater than one.

$\Pi = \Pi_{base} \cup \{1, 2, 7, 12\}$.

Π^* is the set of concrete derivations that both rewrite and yield abstracted concrete phrasal forms.

\mathcal{G} is a **grammatical expansion** of $\hat{\mathcal{G}}$ if and only if both of the following hold:

For every $\langle \hat{A} \rightarrow \hat{\alpha} \rangle \in \hat{P}$, there is a *unique* $\pi \in \Pi$ such that $A \xrightarrow{\pi} \alpha$, $\psi(A) = \hat{A}$, and $\psi(\alpha) = \hat{\alpha}$.

For every $\pi \in \Pi$, where $A \xrightarrow{\pi} \alpha$, either $\psi(A) = \psi(\alpha)$ or $\langle \psi(A) \rightarrow \psi(\alpha) \rangle \in \hat{P}$.

Or, less precisely, \mathcal{G} is a grammatical expansion of $\hat{\mathcal{G}}$ when for each basic concrete derivation from abstracted non-terminal to abstracted phrasal form there is a unique abstract rule, and vice versa.

■

For the remainder of this section, let $\hat{\mathcal{G}}$ and \mathcal{G} be context-free grammars such that \mathcal{G} is an expansion of $\hat{\mathcal{G}}$.

2.2 Contract and Expand

Any concrete derivation that satisfies certain constraints can be converted into a precisely corresponding abstract derivation. Any abstract derivation at all can be converted into an almost precisely corresponding concrete derivation. This concrete derivation satisfies the conversion constraints on concrete derivations. However, the concrete derivation may contain additional rule sequences which correspond to cyclic rules not present in the abstract derivation. This section shows how to obtain these conversions.

2.2.1 Π Subsets

There are concrete derivations in Π that correspond to null abstract derivations rather than to abstract rules.

DEFINITION: Trivial Concrete Derivations

$\Pi_{trivial}$ is the set of concrete derivations in Π that do not correspond to an abstract rule.

Define $\Pi_{trivial} \subseteq \Pi$ by

$$\Pi_{trivial} = \{\pi \in \Pi \mid A \xrightarrow{\pi} \alpha, \text{ where } A \in \mathcal{A}_{chain}, \alpha \in (\Sigma \cup \mathcal{A}_{chain})^*, \text{ and } \psi(A) = \psi(\alpha)\}.$$

For all $\pi \in \Pi_{trivial}$, $A \xrightarrow{\pi} B$, where $A, B \in \mathcal{A}_{chain}$ and $\psi(A) = \psi(B)$.

Note that while $\Pi \setminus \Pi_{base} \subseteq \Pi_{trivial}$, the reverse need not be true.

■

Example:

$$\Pi_{trivial} = \{1, 2, 7, 9, 11, 12\} \not\subseteq \{1, 2, 7, 12\} = \Pi \setminus \Pi_{base}.$$

There are concrete derivations in Π corresponding to cyclic abstract rules.

DEFINITION: Cyclic Concrete Derivations

Π_{cyclic} is the set of concrete derivations that correspond to abstract rules in \hat{P}_{cyclic} .

Define $\Pi_{cyclic} = \{\pi \in \Pi \mid Contract(\pi) \in \hat{P}_{cyclic}\} \subseteq \Pi$.

Note that $\Pi_{cyclic} \cap \Pi_{trivial} = \emptyset$.

■

Example:

$$\Pi_{cyclic} = \{14\}.$$

2.2.2 Contraction

A concrete derivation can be transformed into a corresponding abstract derivation.

DEFINITION: Concrete Derivation Contraction

Define $Contract : \Pi \rightarrow \hat{P} \cup \{\emptyset_A \mid A \in \hat{N}\}$ by

$$\forall \pi \in \Pi, \text{ where } A \xrightarrow{\pi} \alpha, \quad Contract(\pi) = \begin{cases} \emptyset_{\psi(A)} & \text{if } \pi \in \Pi_{trivial} \\ \langle \psi(A) \rightarrow \psi(\alpha) \rangle & \text{otherwise} \end{cases}$$

Extend $Contract$ to $Contract : \Pi^* \rightarrow \hat{P}^*$ by

$$\begin{aligned} Contract(\langle A \xrightarrow{\pi_0} \xi B \zeta \xrightarrow{\pi_1} \xi \beta \zeta \rangle) &= \langle \psi(A) \xrightarrow{Contract(\pi_0)} \psi(\xi B \zeta) \xrightarrow{Contract(\pi_1)} \psi(\xi \beta \zeta) \rangle \\ &\text{and} \\ Contract(\emptyset_A) &= \emptyset_{\psi(A)} \end{aligned}$$

where $\langle A \xrightarrow{\pi_0} \xi B \zeta \rangle, \langle B \xrightarrow{\pi_1} \beta \rangle \in \Pi^*$.

Note that if $\pi \in \Pi^*$ is a concrete derivation such that $A \xRightarrow{\pi} \alpha$, where $A \in \mathcal{A}_{chain}$ and $\alpha \in (\Sigma \cup \mathcal{A}_{chain})^*$,
 $\psi(A) \xRightarrow{Contract(\pi)} \psi(\alpha)$.

■

Example:

π	$Contract(\pi)$
1	$\emptyset \widehat{\text{stmt}}$
2	$\emptyset \widehat{\text{stmt}}$
3	$\hat{1}$
4:5	$\hat{2}$
4:6	$\hat{3}$
7	$\emptyset \widehat{\text{expr}}$
8	$\hat{5}$
9	$\emptyset \widehat{\text{expr}}$
10	$\hat{6}$
11	$\emptyset \widehat{\text{expr}}$
12	$\emptyset \widehat{\text{expr}}$
13	$\hat{7}$
14	$\hat{4}$

2.2.3 Expansion

Any two concrete non-terminals in \mathcal{A} that map onto the same abstract non-terminal \hat{A} can derive each other in \mathcal{G} , possibly with some terminals added, using only trivial and cyclic derivations. This property is needed when transforming an abstract derivation into a corresponding concrete derivation: an abstract non-terminal in a given context may correspond to one concrete non-terminal as the *lhs* of a concrete sub-derivation and as a different concrete non-terminal as part of the *rhs* of a concrete sub-derivation. Thus, the corresponding concrete derivation must include a trivial/cyclic sub-derivation that rewrites the one concrete non-terminal as the other, possibly with some terminals added.

DEFINITION: Concrete Non-Terminal Coercion

Let $B, C \in \mathcal{A}$ such that $\psi(B) = \psi(C)$ be given.

Then there is at least one concrete derivation $\pi \in (\Pi_{trivial} \cup \Pi_{cyclic})^*$ such that $B \xRightarrow{\pi} \xi C \zeta$, where

$\xi, \zeta \in \Sigma^*$ and $Contract(\pi) \in \hat{P}_{cyclic}^*$.

Define $Coerce(B, C) = \pi$, where π is arbitrarily chosen from among the shortest such derivations.

■

Example:

$Coerce(B, C)$		C			
		expr	term	factor	primary
B	expr	\emptyset_{expr}	9	9:11	9:11:12
	term	11:12:14	\emptyset_{term}	11	11:12
	factor	12:14	12:14:9	$\emptyset_{\text{factor}}$	12
	primary	14	14:9	14:9:11	$\emptyset_{\text{primary}}$

Note that even though $\psi(\text{sup-expr}) = \psi(\text{expr})$, $\text{sup-expr} \notin \mathcal{A}$, so $Coerce(\text{sup-expr}, \text{expr})$ is undefined.

An abstract derivation can be transformed into a corresponding concrete derivation. The resulting concrete derivation may have more cyclic derivations than the abstract derivation had cyclic rules, but each rule in the abstract derivation will correspond to a non-trivial and non-cyclic sub-derivation of the concrete derivation, and vice versa.

An abstract rule can be transformed into a corresponding concrete derivation.

DEFINITION: Abstract Rule Expansion

Define *Rule-Expand* : $\hat{P} \rightarrow \Pi^*$ as follows:

Let $\hat{p} \in \hat{P}$ be an abstract rule such that $\hat{A} \xrightarrow{\hat{p}} \hat{\alpha}$, where $\hat{A} \in \hat{N}$ and $\hat{\alpha} \in (\hat{\Sigma} \cup \hat{N})^*$.

By the definition of grammatical expansion, there is a unique concrete derivation $\pi_1 \in \Pi$ such that $A' \xrightarrow{\pi_1} \alpha'$, where $A' \in \mathcal{A}_{\text{chain}}$, $\alpha' \in (\Sigma \cup \mathcal{A}_{\text{chain}})^*$, $\psi(A') = \hat{A}$, and $\psi(\alpha') = \hat{\alpha}$.

In fact, $\pi_1 \in \Pi_{\text{base}} \subseteq \Pi$.

By the definitions of Π_{base} and Π_{trivial} , there are rightmost derivations $\pi_0, \pi_2 \in \Pi_{\text{trivial}}^*$ such that $A \xrightarrow{\pi_0} A' \xrightarrow{\pi_1} \alpha' \xrightarrow{\pi_2} \alpha$, where $A \in \mathcal{A}$, $\alpha \in (\Sigma \cup \mathcal{A})^*$, $\psi(A) = \psi(A')$, and $\psi(\alpha) = \psi(\alpha')$.

By the definition of *Chained* and the unambiguity of \mathcal{G} , π_0 and π_1 are unique.

Set *Rule-Expand*(\hat{p}) = $\pi = \langle A \xrightarrow{\pi_0} A' \xrightarrow{\pi_1} \alpha' \xrightarrow{\pi_2} \alpha \rangle$.

$$\begin{aligned} \text{Contract}(\pi) &= \text{Contract}(\langle A \xrightarrow{\pi_0} A' \xrightarrow{\pi_1} \alpha' \xrightarrow{\pi_2} \alpha \rangle) = \\ &\langle \psi(A) \xrightarrow{\text{Contract}(\pi_0)} \psi(A') \xrightarrow{\text{Contract}(\pi_1)} \psi(\alpha') \xrightarrow{\text{Contract}(\pi_2)} \psi(\alpha) \rangle = \\ &\text{Contract}(\pi_1) = \hat{p}, \text{ since } \pi_0, \pi_2 \in \Pi_{\text{trivial}}^*. \end{aligned}$$

Note that $\text{lhs}(\pi) \in \mathcal{A}$, $\text{rhs}(\pi) \in (\Sigma \cup \mathcal{A})^*$, $\psi(\text{lhs}(\pi)) = \hat{A}$, and $\psi(\text{rhs}(\pi)) = \hat{\alpha}$.

Any abstract derivation can be transformed into a corresponding concrete derivation.

DEFINITION: Abstract Derivation Expansion

Let $\hat{\pi} \in \hat{P}^*$ be an abstract derivation such that $\hat{A} \xrightarrow{\hat{\pi}} \hat{\alpha}$, where $\hat{A} \in \hat{N}$ and $\hat{\alpha} \in (\hat{\Sigma} \cup \hat{N})^*$.

Then there exist one or more derivations $\pi \in \Pi^*$ in \mathcal{G} and $\hat{\pi}' = \text{Contract}(\pi)$ in $\hat{\mathcal{G}}$ such that $A \xrightarrow{\pi} \alpha$ and $\hat{A} \xrightarrow{\hat{\pi}'} \hat{\alpha}'$, where $A \in \mathcal{A}$, $\alpha \in (\Sigma \cup \mathcal{A})^*$, $\hat{\alpha}' \in (\hat{\Sigma} \cup \hat{N})^*$, $\psi(A) = \hat{A}$, $\hat{\alpha}' = \psi(\alpha)$, and

$$\hat{\pi} \wedge \hat{P}_{\text{cyclic}} = \hat{\pi}' \wedge \hat{P}_{\text{cyclic}}.$$

$Expand(\hat{\pi})$ will be defined as the derivation π such that π is any of the shortest derivations that satisfies the given conditions.

$Expand$ is constructed inductively on the length n of $\hat{\pi}$.

Base case ($n = 0$):

Define $Expand(\emptyset_{\hat{A}}) = \emptyset_A$, where $A \in ((\psi \cap (\mathcal{A} \times \hat{N}))^{-1}(\hat{A}))$.

If there is more than one such A , $Expand(\emptyset_{\hat{A}})$ is non-deterministically defined as each of them.

The claim is trivially true.

Secondary base case ($n = 1$):

Let $\hat{p} \in \hat{P}$ be an abstract derivation of length 1 such that $\hat{A} \xrightarrow{\hat{p}} \hat{\alpha}$, where $\hat{A} \in \hat{N}$ and $\hat{\alpha} \in (\hat{\Sigma} \cup \hat{N})^*$.

Set $Expand(\hat{p}) = Rule-Expand(\hat{p})$.

The claim follows from the definition of $Rule-Expand$.

Inductive case ($n > 1$):

Let $\hat{\pi} \in \hat{P}^*$ be an abstract derivation of length n such that $\hat{A} \xrightarrow{\hat{\pi}} \hat{\alpha}$, where $\hat{A} \in \hat{N}$ and $\hat{\alpha} \in (\hat{\Sigma} \cup \hat{N})^*$.

Let $\langle \hat{A} \xrightarrow{\hat{\pi}} \hat{\alpha} \rangle = \langle \hat{A} \xrightarrow{\hat{\pi}_0} \hat{\gamma} \hat{B} \hat{\delta} \xrightarrow{\hat{p}} \hat{\gamma} \hat{\beta} \hat{\delta} \rangle$, where $\hat{\pi}_0 \in \hat{P}^*$, $\hat{p} \in \hat{P}$, $\hat{B} \in \hat{N}$, and $\hat{\gamma}, \hat{\delta}, \hat{\beta} \in (\hat{\Sigma} \cup \hat{N})^*$.

By the inductive hypothesis, there are derivations $\pi_0 \in \Pi^*$ in \mathcal{G} and $\hat{\pi}'_0 = Contract(\pi_0)$ in $\hat{\mathcal{G}}$ such that $A \xrightarrow{\pi_0} \gamma B \delta$ and $\hat{A} \xrightarrow{\hat{\pi}'_0} \hat{\gamma}' \hat{B} \hat{\delta}'$, where $A, B \in \mathcal{A}$, $\gamma, \delta \in (\Sigma \cup \mathcal{A})^*$, $\hat{\gamma}', \hat{\delta}' \in (\hat{\Sigma} \cup \hat{N})^*$, $\psi(A) = \hat{A}$, $\hat{\gamma}' \hat{B} \hat{\delta}' = \psi(\gamma B \delta)$, and $\hat{\pi}'_0 \setminus \hat{P}_{cyclic} = \hat{\pi}_0 \setminus \hat{P}_{cyclic}$.

Define $\pi_2 \in \Pi^*$ by $\pi_2 = Rule-Expand(\hat{p})$.

By the definition of $Rule-Expand$, $C \xrightarrow{\pi_2} \beta$, where $C \in \mathcal{A}$, $\beta \in (\Sigma \cup \mathcal{A})^*$, $\psi(C) = \hat{B}$, and $\psi(\beta) = \hat{\beta}$.

Define the concrete derivation $\pi_1 = Coerce(B, C) \in (\Pi_{trivial} \cup \Pi_{cyclic})^*$.

By the definition of $Coerce$, $B \xrightarrow{\pi_1} \xi C \zeta$, where $\xi, \zeta \in \Sigma^*$ and $\hat{\pi}_1 = Contract(\pi_1) \in \hat{P}_{cyclic}^*$.

Define $\pi = \langle A \xrightarrow{\pi_0} \gamma B \delta \xrightarrow{\pi_1} \gamma \xi C \zeta \delta \xrightarrow{\pi_2} \gamma \xi \beta \zeta \delta \rangle$ and $\alpha = \gamma \xi \beta \zeta \delta \in (\Sigma \cup \mathcal{A})^*$.

Define $\hat{\pi}' = Contract(\pi) = Contract(\langle A \xrightarrow{\pi_0} \gamma B \delta \xrightarrow{\pi_1} \gamma \xi C \zeta \delta \xrightarrow{\pi_2} \gamma \xi \beta \zeta \delta \rangle) =$

$$\begin{aligned} & \langle \hat{A} \xrightarrow{Contract(\pi_0)} \hat{\gamma} \hat{B} \hat{\delta} \xrightarrow{Contract(\pi_1)} \hat{\gamma} \hat{\xi} \hat{B} \hat{\zeta} \hat{\delta} \xrightarrow{Contract(\pi_2)} \hat{\gamma} \hat{\xi} \hat{\beta} \hat{\zeta} \hat{\delta} \rangle = \\ & \langle \hat{A} \xrightarrow{\hat{\pi}'_0} \hat{\gamma} \hat{B} \hat{\delta} \xrightarrow{\hat{\pi}_1} \hat{\gamma} \hat{\xi} \hat{B} \hat{\zeta} \hat{\delta} \xrightarrow{\hat{p}} \hat{\gamma} \hat{\xi} \hat{\beta} \hat{\zeta} \hat{\delta} \rangle. \end{aligned}$$

Define $\hat{\alpha}' = \psi(\alpha) = \psi(\gamma \xi \beta \zeta \delta) = \hat{\gamma} \hat{\xi} \hat{B} \hat{\zeta} \hat{\delta} \in (\hat{\Sigma} \cup \hat{N})^*$.

Clearly $A \xrightarrow{\pi} \alpha$ and $\hat{A} \xrightarrow{\hat{\pi}'} \hat{\alpha}'$.

$\hat{\pi} \setminus \hat{P}_{cyclic} = \langle \hat{A} \xrightarrow{\hat{\pi}_0} \hat{\gamma} \hat{B} \hat{\delta} \xrightarrow{\hat{\pi}_1} \hat{\gamma} \hat{\xi} \hat{B} \hat{\zeta} \hat{\delta} \xrightarrow{\hat{p}} \hat{\gamma} \hat{\xi} \hat{\beta} \hat{\zeta} \hat{\delta} \rangle \setminus \hat{P}_{cyclic} = \langle \hat{A} \xrightarrow{\hat{\pi}_0} \hat{\gamma} \hat{B} \hat{\delta} \xrightarrow{\hat{p}} \hat{\gamma} \hat{\beta} \hat{\delta} \rangle \setminus \hat{P}_{cyclic} = \hat{\pi} \setminus \hat{P}_{cyclic}$.

Define $Expand(\hat{\pi}) = \pi$.

This completes the induction, and thus the construction. ■

2.2.4 Inversion

Contract and *Expand* are very nearly inverses: *Contract* is the right inverse of *Expand*, and is *nearly* the left inverse as well. The mapping (*Contract* \circ *Expand*) is virtually an identity mapping; it may add a few cyclic rules but does not otherwise modify derivations.

THEOREM: Invertibility of *Contract* and *Expand*

Let $\hat{\pi} \in \hat{P}^*$ be an abstract derivation such that $\hat{A} \xRightarrow{\hat{\pi}} \hat{\alpha}$, where $\hat{A} \in \hat{N}$ and $\hat{\alpha} \in (\hat{\Sigma} \cup \hat{N})^*$.

Define the abstract derivation $\hat{\pi}' \in \hat{P}^*$ by $\hat{\pi}' = \text{Contract}(\text{Expand}(\hat{\pi}))$.

Then $\hat{A} \xRightarrow{\hat{\pi}'} \hat{\alpha}'$, where $\hat{\alpha}' \in (\hat{\Sigma} \cup \hat{N})^*$ and $\hat{\pi}' \setminus \hat{P}_{\text{cyclic}} = \hat{\pi} \setminus \hat{P}_{\text{cyclic}}$.

Let $\pi \in \Pi^*$ be a concrete derivation such that $A \xRightarrow{\pi} \alpha$, where $A \in \mathcal{A}_{\text{chain}}$ and $\alpha \in (\Sigma \cup \mathcal{A}_{\text{chain}})^*$.

Define the concrete derivation $\pi' \in \Pi^*$ by $\pi' = \text{Expand}(\text{Contract}(\pi))$.

Then $A' \xRightarrow{\pi'} \alpha'$, where $A' \in \mathcal{A}$, $\alpha' \in (\Sigma \cup \mathcal{A})^*$, $\psi(A') = \psi(A)$, and $\psi(\alpha') = \psi(\alpha)$.

If $A \in \mathcal{A}$ then $A' = A$, if $\alpha \in (\Sigma \cup \mathcal{A})^*$ then $\alpha' = \alpha$, and if $A \in \mathcal{A}$ and $\alpha \in (\Sigma \cup \mathcal{A})^*$ then $\pi' = \pi$.

If $\pi' = \emptyset_{A'}$, then $\pi' = \text{Expand}(\emptyset_{\psi(A)})$ is non-deterministically defined, and $\pi' = \emptyset_{\text{Cycled}(\text{Chained}(A))}$ is the only choice for which the preceding claim holds true.

■

2.3 Simplifications

The definition of grammatical abstraction permits a set of concrete non-terminals to be cycle equivalent in very complicated ways. The members of a concrete non-terminal cycle equivalence class can relate to each other in arbitrarily complex cycles involving any number of cyclic concrete derivations.

Example:

To illustrate the sort of strange cycles that may exist, consider adding both the concrete rule $\langle \text{primary} \rightarrow [\text{expr}] \rangle$ and the corresponding abstract rule $\langle \widehat{\text{expr}} \rightarrow \widehat{[\text{expr}]} \rangle$ to the example grammars. Further add the rule $\langle \widehat{\text{expr}} \rightarrow \widehat{[\text{expr}]} \rangle$ to \hat{P}_{cyclic} . Although \mathcal{G} is still a grammatical expansion of $\hat{\mathcal{G}}$, these rules adds an extra link in the expr cyclic equivalence set. One consequence of this would be that $\text{Coerce}(\text{primary}, \text{expr})$ could be either $\text{primary} \xRightarrow{*} [\text{expr}]$ or $\text{primary} \xRightarrow{*} (\text{expr})$.

However, the concept of grammatical expansion can be applied more easily if there is at most one cyclic concrete derivation for each concrete non-terminal cycle equivalence class. This section adds this requirement and explores the consequences.

RESTRICTION: At Most One Cyclic Rule For Each Abstract Non-Terminal

With only a single cyclic rule for an abstract non-terminal, there can be only a single cyclic con-

crete derivation for the corresponding concrete non-terminal cycle equivalence class. Further, if this derivation is divided into sub-derivations in Π , all but one of the sub-derivations will be in $\Pi_{trivial}$. This has implications for the concrete representation of abstract non-terminals and for the determinism of *Coerce*.

For all $\hat{p}, \hat{q} \in \hat{P}_{cyclic}$ if $lhs(\hat{p}) = lhs(\hat{q})$, then $\hat{p} = \hat{q}$.

Example:

The restriction is true of the given \hat{P}_{cyclic} .

Again consider adding both the concrete rule $\langle primary \rightarrow [expr] \rangle$ and the corresponding abstract rule $\langle \widehat{expr} \rightarrow [\widehat{expr}] \rangle$ to the example grammars.

Whether or not this new abstract rule is added to \hat{P}_{cyclic} , \mathcal{G} is still a grammatical expansion of $\hat{\mathcal{G}}$.

Note that \hat{P}_{cyclic} must contain at least one of the abstract rules $\langle \widehat{primary} \rightarrow [\widehat{expr}] \rangle$ and $\langle \widehat{expr} \rightarrow [\widehat{expr}] \rangle$ for \mathcal{G} to be a grammatical expansion of $\hat{\mathcal{G}}$ relative to \hat{P}_{cyclic} .

However, because of the restriction, \hat{P}_{cyclic} may contain *either* this new abstract rule *or* the original abstract rule $\langle \widehat{expr} \rightarrow [\widehat{expr}] \rangle$, but *not* both.

The result of applying *Expand* to a zero-length abstract derivation may be non-deterministic. Further, the concrete context in which the expansion occurs may require some property of the result which is true of some but not all of the possible results.

For example, non-deterministically define the concrete non-terminal $A = lhs(Expand(\emptyset, \widehat{expr}))$.

Sometimes it is necessary that $A \xRightarrow{*} term * factor$, which is true when $A \in \{expr, term\}$.

Other times it is necessary that $expr \xRightarrow{*} term * A$, which is true when $A \in \{factor, primary\}$.

As illustrated by this example, it is not possible to produce a single result which will satisfy all possible contexts. However, it is possible to produce one result that will satisfy all possible contexts where the non-terminal appears in the *lhs* of a derivation, as in the first case, and another result that will satisfy all possible contexts where the non-terminal appears in the *rhs* of a derivation, as in the second case.

For each abstract non-terminal \hat{A} there is a unique concrete non-terminal $Top(\hat{A})$ which derives any concrete non-terminal in \mathcal{A} that maps onto \hat{A} . This top non-terminal may always be used as the concrete non-terminal corresponding to \hat{A} when \hat{A} is the *lhs* of an abstract derivation. The name *Top* is used because the *lhs* of an abstract derivation is represented by the top of the corresponding syntax tree.

DEFINITION: Abstract Non-Terminal Top Concrete Representation

Define $\mathcal{A}_{top} \subseteq \mathcal{A}$ by

$$\mathcal{A}_{top} = \{A \in \mathcal{A} \mid \neg \exists \pi \in \Pi_{trivial}^*, B \xRightarrow{\pi} A \text{ where } B \in \mathcal{A} \text{ and } B \neq A\}.$$

$\psi \cap (\mathcal{A}_{top} \times \hat{N})$ is invertible.

Define $Top : (\hat{\Sigma} \cup \hat{N}) \rightarrow (\Sigma \cup \mathcal{A}_{top})$ by $Top = (\psi \cap ((\Sigma \cup \mathcal{A}_{top}) \times (\hat{\Sigma} \cup \hat{N})))^{-1}$.

For all $\hat{A} \in \hat{N}$, $Top(\hat{A}) \xrightarrow[\Pi_{trivial}]{*} lhs(Expand(\emptyset_{\hat{A}}))$.

Example:

$\mathcal{A}_{top} = \{\text{stmt}, \text{expr}\}$.
 $Top(\widehat{\text{stmt}}) = \text{stmt}$ and $Top(\widehat{\text{expr}}) = \text{expr}$.

For each abstract non-terminal \hat{A} there is a unique concrete non-terminal $Bottom(\hat{A})$ which can be derived from any concrete non-terminal in \mathcal{A} that maps onto \hat{A} . This bottom non-terminal may always be used as the concrete non-terminal corresponding to \hat{A} when \hat{A} is in the *rhs* of an abstract derivation. The name *Bottom* is used because the *rhs* of an abstract derivation is represented by the bottom or frontier of the corresponding syntax tree.

DEFINITION: Abstract Non-Terminal Bottom Concrete Representation

Define $\mathcal{A}_{bottom} \subseteq \mathcal{A}$ by

$$\mathcal{A}_{bottom} = \{A \in \mathcal{A} \mid \neg \exists \pi \in \Pi_{trivial}^*, A \xrightarrow{\pi} B \text{ where } B \in \mathcal{A} \text{ and } B \neq A\}.$$

$\psi \cap (\mathcal{A}_{bottom} \times \hat{N})$ is invertible.

Define $Bottom : (\hat{\Sigma} \cup \hat{N}) \rightarrow (\Sigma \cup \mathcal{A}_{bottom})$ by $Bottom = (\psi \cap ((\Sigma \cup \mathcal{A}_{bottom}) \times (\hat{\Sigma} \cup \hat{N})))^{-1}$.

For all $\hat{A} \in \hat{N}$, $lhs(Expand(\emptyset_{\hat{A}})) \xrightarrow[\Pi_{trivial}]{*} Bottom(\hat{A})$.

Example:

$\mathcal{A}_{bottom} = \{\text{stmt}, \text{primary}\}$.
 $Bottom(\widehat{\text{stmt}}) = \text{stmt}$ and $Bottom(\widehat{\text{expr}}) = \text{primary}$.

For any abstract non-terminal $\hat{A} \in \hat{N}$, the concrete non-terminal $Top(\hat{A})$ (or $Bottom(\hat{A})$) may always be used for $lhs(Expand(\emptyset_{\hat{A}}))$ when it is known that the concrete non-terminal will be used in the *lhs* (or *rhs*) of a concrete derivation, as is the case for top-down syntactic elaboration (or bottom-up incremental parsing).

The cyclic rules specified by the restricted \hat{P}_{cyclic} determine a unique *Coerce* mapping.

DEFINITION: Concrete Non-Terminal Coercion (Unique)

With the restriction placed on \hat{P}_{cyclic} , there is only one *Coerce* mapping.
There is a unique minimal partial order $<$ on \mathcal{A} such that

$$\text{for all } \hat{A} \in \hat{N}, \text{ for all } B, C \in (\psi \cap (\mathcal{A} \times \hat{N}))^{-1}(\hat{A}), (Coerce(B, C) \notin \Pi_{trivial}^* \Leftrightarrow C < B).$$

There is an efficient algorithm to compute $Contract \circ Coerce$:

$$\forall B, C \in \mathcal{A}, \text{ where } \psi(B) = \psi(C),$$

$$Contract(Coerce(B, C)) = \begin{cases} \hat{p} \in \hat{P}_{cyclic}, \text{ where } lhs(\hat{p}) = \psi(B) & \text{if } C \prec B \\ \emptyset_{\psi(B)} & \text{otherwise.} \end{cases}$$

■

Example:

$expr \prec term \prec factor \prec primary$.

2.4 Contract and Expand Algorithms

The *Contract* and *Expand* mappings convert between abstract and concrete derivations. An abstract derivation is represented by an abstract syntax tree. A concrete derivation can be similarly represented by a concrete syntax tree. Thus algorithms for the *Contract* and *Expand* mappings apply to abstract and concrete syntax trees. A *Contract* algorithm takes a concrete syntax tree and produces a corresponding abstract syntax tree, and an *Expand* algorithm does the reverse.

The basic algorithms for *Contract* and *Expand* are identical. The abstract [concrete] syntax tree is partitioned into a set of sub-trees that represent abstract [concrete] derivations in $P[\Pi]$. By the definition of grammatical expansion, each such abstract [concrete] derivation corresponds to a unique concrete [abstract] derivation in $\Pi[P]$; a concrete [abstract] sub-tree is constructed to represent this corresponding derivation. The resulting concrete [abstract] sub-trees are then combined into a single concrete [abstract] tree in the same way that the set of abstract [concrete] sub-trees formed the original abstract [concrete] tree.

The algorithm just described has five steps:

1. Partition the syntax tree.
2. Determine the derivation represented by each sub-tree.
3. Determine the other grammar derivation corresponding to each represented derivation.
4. Construct a sub-tree for each such corresponding derivation.
5. Combine these sub-trees into a single tree.

The first step is easily done for *Expand* by dividing the tree into unit sub-trees, and for *Contract* by dividing the tree at nodes that represent symbols in \mathcal{A}_{chain} . For *Expand*, the second step is trivial, since each unit sub-tree designates the rule (or derivation) that the sub-tree represents. The third step can be handled by a simple table lookup. The fourth step is a straightforward tree operation. For *Contract*, the fifth step is also a straightforward tree operation. However, the fifth step for *Expand* and the second step for *Contract* are not so simple.

Syntax trees are combined by appending one tree to another at a leaf of the latter tree, where the non-terminal R represented by the former tree's root is the same as L , the non-terminal represented

by the latter tree's leaf. However, the *Expand* algorithm needs to combine trees for which R and L are not necessarily the same, although they are guaranteed to be cycle equivalent. This problem is handled by constructing a third syntax tree whose root represents L and whose unique non-terminal leaf represents R . Such a tree can be constructed by applying steps three and four to the derivation $Contract(Coerce(L, R))$, which is efficiently computable using the results of the previous section. With this third syntax tree interposed between the other two, it is possible to construct a single syntax tree that combines the two trees.

Determining the Π derivation represented by a concrete syntax tree can be done in a bottom-up manner as follows: For each sub-tree whose leaves are leaves of the whole tree, generate a state representing the sub-derivations it matches, called the *Contract* state. The set of *Contract* states is Π_{sub} , the (rightmost) sub-derivations of Π , defined by

$$\Pi_{sub} = \left\{ \pi \in P^* \mid \begin{array}{l} \langle A \xRightarrow{*} \xi B \zeta \xRightarrow{\pi} \xi \beta \zeta \rangle \in \Pi, \text{ where } A, B \in \mathcal{A}_{chain}, \\ \xi, \zeta, \beta \in (\Sigma \cup \mathcal{A}_{chain})^*, \text{ and } \pi \text{ is rightmost} \end{array} \right\}.$$

Note that $\Pi \subset \Pi_{sub}$, so Π_{sub} contains a unique state for each derivation in Π , as well as for each sub-derivation. Like Π , Π_{sub} can be infinite in size, but its derivations have a finite number of distinct *lhs* and *rhs*.

Example:

$$\Pi_{sub} = \{5, 6, 1, 2, 7, 12, 3, 4:5, 4:6, 8, 9, 10, 11, 13, 14\}.$$

Note that $\Pi_{sub} \setminus \Pi = \{5, 6\}$, since there are only a couple of derivations in Π with length greater than one.

The *Contract* state of a tree node representing a rule p does not necessarily depend on the state of each of the node's children. For example, the state of a child node representing a terminal $a \in \Sigma$ or a non-terminal $A \in \mathcal{A}_{chain}$ is irrelevant, since such a node will always have exactly the same state (\emptyset_a or \emptyset_A). Similarly, the state of a child node representing a non-terminal from which only one derivation in Π_{sub} begins is also always the same, by definition. Define $N_{ambig} \subset N$ by

$$N_{ambig} = \{A \in N \setminus \mathcal{A}_{chain} \mid \pi, \pi' \in \Pi_{sub}, A \xRightarrow{\pi} \alpha, A \xRightarrow{\pi'} \alpha', \text{ where } \alpha, \alpha' \in (\Sigma \cup \mathcal{A}_{chain})^*, \text{ and } \alpha \neq \alpha'\}.$$

N_{ambig} is the set of non-abstracted non-terminals such that any node representing a member of this set has more than one possible state, depending on the states of its child nodes.

Example:

$N_{ambig} = \{\text{else-clause}\}$, since else-clause is the only concrete non-terminal that is not abstracted, and it is the *lhs* of more than one rule.

The *Contract* state of a tree node representing rule p depends only on the state of child nodes that correspond to symbols of N_{ambig} in the $rhs(p)$. If there are no such symbols in $rhs(p)$, the node's state does not depend on the state of its children at all. Conversely, when the $rhs(p)$ does contain symbols in N_{ambig} , the node's state depends on the state of the corresponding child nodes. Define $P_{ambig} \subset P$ by

$$P_{ambig} = \{p \in P \mid rhs(p) = \xi A \zeta, A \in N_{ambig}, \text{ and } \xi, \zeta \in (\Sigma \cup \mathcal{A}_{chain})^*\}.$$

P_{ambig} is the set of rules such that the state of any node representing a member of this set depends on the state of some of that node's children.

Example:

$P_{ambig} = \{4\}$, since only concrete rule 4 contains the concrete non-terminal else-clause, which is the only element of N_{ambig} .

The *Contract* state for any tree node representing a rule not in P_{ambig} can be computed from the rule itself. However, even this computation is unnecessary if the node's state will never be examined. A node's state is examined only when the node represents a non-terminal in N_{ambig} or the node is the root of a sub-tree representing a derivation in Π . In the latter case, the node also represents a non-terminal in \mathcal{A}_{chain} . Thus, $\mathcal{A}_{chain} \cup N_{ambig}$ is the set of non-terminals such that a state must be computed for any node representing a member of this set.

Each concrete rule $p \in P$ is assigned an abstraction action $action(p)$ describing the computation that must be performed to determine the *Contract* state for any node representing p .

Define $action : P \rightarrow \{\text{none}, \text{mark}, \text{disambiguate}\}$ by

$$\forall p \in P, \quad action(p) = \begin{cases} \text{mark} & \text{if } p \notin P_{ambig} \text{ and } lhs(p) \in (\mathcal{A}_{chain} \cup N_{ambig}) \\ \text{disambiguate} & \text{if } p \in P_{ambig} \\ \text{none} & \text{otherwise.} \end{cases}$$

Mark computes a node's state without examining its children. **Disambiguate** computes a node's state by examining the states of at least some of a node's children. **None** naturally does nothing.

Example:

p	$action(p)$
1	mark (none)
2	mark (none)
3	mark
4	disambiguate
5	mark
6	mark
7	mark (none)
8	mark
9	mark (none)
10	mark
11	mark (none)
12	mark (none)
13	mark
14	mark

Note that some of the **mark** entries have **none** in parentheses. Such rules mark derivations in $\Pi_{trivial}$, which are typically irrelevant to the construction of an abstract syntax tree.

Each concrete rule p also has some data $data(p)$ used in the state computation.

For $action(p) = \text{none}$, $data(p) = \perp$.

When $action(p) = \text{mark}$, $data(p) = \pi \in \Pi_{sub}$, where π is the unique concrete derivation in Π_{sub} that begins with p .

If $action(p) = \text{disambiguate}$, $data(p)$ is a set of pairs $\{(\langle \pi_1, \dots, \pi_n \rangle, \pi)\}$. The first element of each pair, $\langle \pi_1, \dots, \pi_n \rangle$, is a tuple of possible *Contract* states for tree nodes representing the N_{ambig} non-terminals in p 's *rhs*. The second element of each pair, $\pi \in \Pi_{sub}$, is the state of a node representing p whose N_{ambig} children have those states $\pi_1 \dots \pi_n$. $data(p)$ contains all such possible pairs.

Example:

p	$data(p)$
1	1
2	2
3	3
4	$\{\langle 5, 4: 5 \rangle, \langle 6, 4: 6 \rangle\}$
5	5
6	6
7	7
8	8
9	9
10	10
11	11
12	12
13	13
14	14

Note that when Π_{sub} is infinite in size, the *data* mapping as just described maps some rules onto infinite sets. This problem can be eliminated by arbitrarily choosing $\Pi_{sub}' \subseteq \Pi_{sub}$ such that for each derivation $\pi \in \Pi_{sub}$ there is a unique representative derivation $\pi' \in \Pi_{sub}'$ such that $lhs(\pi) = lhs(\pi')$ and $rhs(\pi) = rhs(\pi')$. Since the derivations of Π_{sub} have a finite number of distinct *lhs* and *rhs*, there is at least one such finite Π_{sub}' . The definition of the *data* mapping is then modified by substituting each *Contract* state $\pi' \in \Pi_{sub}'$ for all of the *Contract* states $\pi \in \Pi_{sub}'$ that it represents. This modification yields a *data* mapping that does not map any rule onto an infinite set.

Given a concrete syntax tree representing a derivation in Π , the *action* and *data* mappings can be used to determine the abstract rule or null derivation to which the concrete derivation corresponds. First a *Contract* state is computed for the root of the tree, bottom-up. The operation at a tree node representing the rule p is simple:

If $action(p)$ is **none**, no *Contract* state need be computed.

If $action(p)$ is **mark**, the *Contract* state for the rule is $data(p)$.

If $action(p)$ is **disambiguate**, the *Contract* state is determined by matching the *Contract* states of the node's children to those in the state tuples in the pairs of $data(p)$.

Once the *Contract* state for the tree's root has been computed, the corresponding abstract rule or

null derivation can be looked up in a table. In no case is any tree node examined more than once, and often whole portions of the tree are not examined at all.

3 Input

A Ladle input specification provides the following four things for a language:

- a set of textual expressions for the lexemes
- an abstract extended context-free grammar for the language
- an internal representation for syntax trees of the language
- an LALR(1) concrete extended context-free grammar for the language

Every Ladle description has the form:

```

“LANGUAGE” <identifier>
“LEXICAL”
    <lexical-definition>*
“ABSTRACT”
    <abstract-definition>*
“CONCRETE”
    <concrete-definition>*

```

The *identifier* names the language. Each set of definitions describes the appropriate feature of the language. The abstract section defines both the abstract syntax and the syntax tree IR. The concrete section may be omitted. Appendix B gives both an example and a description of a Ladle description.

A Ladle identifier must begin with an alphabetic character, and may contain any number of alphabetic, numeric, or underscore (“_”) characters. Integers are decimal and unsigned. The keywords of Ladle are case insensitive. Newlines, tabs, and formfeeds between Ladle tokens are whitespace.

A Ladle description may contain comments anywhere that whitespace is legal. A Ladle comment consists of any amount of text between “/*” and “*/”. Comments may be nested.

The Ladle processor requires that the concrete grammar be a grammatical expansion of the abstract grammar. In addition, it imposes the following requirement on a language specification:

For all $\pi, \pi' \in \Pi_{cyclic}$ such that $A \xRightarrow{\pi} \alpha$ and $B \xRightarrow{\pi'} \beta$,
 where $A, B \in \mathcal{A}_{chain}$ and $\alpha, \beta \in (\Sigma \cup \mathcal{A}_{chain})$,
 if $\psi(A) = \psi(B)$ then
 $A = B$ and there is a $C \in \mathcal{A}_{chain}$ such that $\alpha, \beta \in \Sigma^*\{C\}\Sigma^*$.

It is an error for Ladle input not to satisfy this condition. Typical Ladle descriptions satisfy the requirement anyway, so it is not an issue in practice. The requirement simplifies the theory somewhat, as discussed in Section 2.3.

3.1 The Lexical Section

The lexical section of the language specification defines the lexemes for the language. The lexemes are the terminals for both the abstract and concrete grammars, plus lexical constructs such as whitespace and comments that are not grammar terminals. A single set of terminals is used for both the abstract and concrete grammars, so $\Sigma = \hat{\Sigma}$ and $\psi_0 \cap (\Sigma \times \hat{\Sigma}) = 1_{\hat{\Sigma}}$. A *lexical-definition* has the form:

$\langle identifier \rangle \text{ ``=" } \langle lex-expression \rangle \text{ ``=> } \langle lex-status \rangle \text{ ``,"}$

The *identifier* names the lexeme. Each lexeme must have a unique name.

The *lex-expression* is an extended regular expression describing the set of strings that are instances of the lexeme. The Ladle processor combines all of these expressions into an automaton that scans text a character at a time until the longest possible lexeme has been recognized. This automaton looks only one character ahead, and never backs up. If one lexeme is a prefix of another, the automaton will optimistically expect the longer lexeme if the lookahead character is appropriate. When the portion of text that is currently being scanned doesn't match any lexeme, a special error lexeme is recognized.

The basic elements of a lexical expression are strings, case insensitive strings, and character sets. A string is one or more characters delimited by ""s. A case insensitive string is the same, but delimited by ``; an alphabetic character in the string matches either case of the character. The word *string* is normally used to refer to both regular and case insensitive strings. A character set is one or more characters delimited by "{" and "}", and represents the set of characters specified. The "-" character in a character set indicates a range of characters; e.g., "{a-zA-Z}" is the set of alphabetic characters. The "\" character is a quoting character in strings and character sets, interpreted as follows:

"\"	backslash
"\"	hyphen
"\n"	newline
"\t"	tab
"\d"	delete
"\b"	backspace
"\e"	escape
"\^X"	control-X
"\n"	ascii n, octal

No other quoting convention is supported; in particular "\X" is illegal in the general case. It is also illegal to use "\" anywhere else, for example, in an identifier.

The basic lexical operators, in order of precedence from highest to lowest, are:

$\langle expr \rangle *$	at least 0 repetitions of <i>expr</i>
$\langle expr \rangle +$	at least 1 repetition of <i>expr</i>
$\langle expr \rangle \langle expr \rangle$	concatenation
$\langle expr \rangle \langle expr \rangle$	alternation

$[\langle expr \rangle]$	optional
$(\langle expr \rangle)$	grouping

There are also some extended lexical operators whose use is more restricted. They are:

$\langle expr \rangle - \langle string \rangle$	string match
$\langle string \rangle \sim \langle string \rangle$	balanced string match
$\langle string \rangle \text{ IN } \langle identifier \rangle$	keyword

The *string* operands of the match operators “-” and “~” must not be case insensitive strings. Both of these operators match strings that begin with the left operand and end with the right operand. The balanced version only matches strings with balanced occurrences of the left and right operands. The match operators must not be nested within either of the repetition operators. Also, if a match operator appears in an operand of the concatenation operator, it must be in the final operand. The keyword operator “IN” is useful for defining special instances of other lexical patterns that are to be treated as distinct lexemes. The right operand must be the name of a previously defined lexeme. The left operand may be any string that matches the expression for this previously defined lexeme. The keyword operator may only be nested within the alternation or grouping operators. Appendix B includes examples of each of these specialized operators.

Any lexeme whose expression operators are all keyword, concatenation, or grouping operators is a *constant lexeme*. Such a lexeme matches only a single string, although parts of this string may be case insensitive.

Not all lexemes are terminal symbols for the language’s grammars, nor are all lexemes included in the IR tree. The status of a lexeme is described by its *lex-status*, which must be one of

IGNORE
SCREEN
OMIT
PRESERVE

or may be left out, along with the “=>” preceding it. An ignored lexeme is not a grammar terminal, nor is it ever in an IR tree. A screened lexeme is also not a grammar terminal, but it should be included in IR trees as an annotation of some sort. Omitted and preserved lexemes are grammar terminals, and may or may not be found in IR trees. A preserved lexeme is by default represented explicitly, while an omitted one is by default represented implicitly. In either case, the default can be overridden for a particular instance of the lexeme in the abstract grammar. A warning is issued when a non-constant lexeme is omitted. If no lexeme status is given, the default status assumed is OMIT for constant lexemes, PRESERVE for all others. Appendix B includes examples of lexeme status. Typically status IGNORE is used for whitespace, SCREEN for comment lexemes, OMIT for constant lexemes such as keywords, and PRESERVE for non-constant lexemes, such as identifiers.

Outside of the lexical section, a lexeme may be referenced by its name, or by a string. In the latter case, if there is not already a lexeme defined as that string, a new lexeme is so defined with lexical status OMIT. The abstract and concrete sections must refer only to the terminal lexemes, which are the lexemes that are neither ignored nor screened.

3.2 Extended Grammar Notation

Each of the abstract and concrete sections of the language description contains an extended context-free grammar. Such a grammar consists of a set of non-terminals each of which has some rewrite rules. The *rhs* of a rewrite rule may be any of these forms:³

<i>symbols</i>	conventional <i>rhs</i>
<i>[symbols]</i>	optional
<i>symbols *</i>	at least 0 repetitions of <i>symbols</i>
<i>symbols * lexeme</i>	the same, but repetitions separated by <i>lexeme</i>
<i>symbols +</i>	at least 1 repetition of <i>symbols</i>
<i>symbols + lexeme</i>	the same, but repetitions separated by <i>lexeme</i>
<i>symbols ++</i>	at least 2 repetitions of <i>symbols</i>
<i>symbols ++ lexeme</i>	the same, but repetitions separated by <i>lexeme</i>

Symbols is any non-empty sequence of grammar symbols, although in a conventional *rhs* it may be empty. The optional operator is really just a short hand notation: the rule $A \rightarrow [\alpha]$ represents the rules $A \rightarrow \alpha$ and $A \rightarrow \epsilon$. Any rule containing one of the repetition operators is called a *sequence* rule. The lexeme separating successive repetitions is called a *delimiter*. A warning is issued when a delimiter is not a constant lexeme.

The “++” sequence operator is not standard usage, although the other extended grammar operators are. The Ladle description of Ladle in Appendix B contains an example of its use to describe lexical concatenation. The concatenation of several lexical expressions is properly represented as a sequence. However, a single expression should not be represented as a sequence of one expression, but simply as the expression itself. The “++” operator makes this distinction possible.

Neither grammar expansion nor LALR(k) are defined on extended grammars, but both definitions can be generalized. Appendix C describes how to normalize an extended grammar into a conventional one. Similarly, with respect to a given ψ_0 and \hat{P}_{cyclic} , an extended grammar \mathcal{G} is said to be an expansion of another extended grammar $\hat{\mathcal{G}}$ if and only if the the normalization of \mathcal{G} is an expansion of the normalization of $\hat{\mathcal{G}}$. In practical terms, this means that the sequence rules of the two grammars must correspond in the same way as other rules, with the sequence operators considered lexemes. An extended grammar \mathcal{G} is LALR(k) if and only if the normalization of \mathcal{G} is LALR(k).

3.3 The Abstract Section

The abstract section of the language specification describes both the language’s abstract syntax and the internal representation of syntax trees of the language. The abstract syntax is described as a possibly ambiguous context-free grammar $\hat{\mathcal{G}}$. Each of the definitions in the abstract section defines a non-terminal of \hat{N} . An *abstract-definition* has the form:

³In the current implementation of Ladle, a non-terminal which has an optional or sequence rule may only have a single rule. This was a poor design choice, as Ladle itself is best described by violating this restriction, as in Appendix B.

$\langle identifier \rangle = \langle abstract-rule-seq \rangle ;$

The *identifier* names the non-terminal. Each non-terminal must have a unique name, which must not be the name of any lexeme. The first abstract non-terminal defined is the start symbol \hat{S} for the abstract grammar.

An *abstract-rule-seq* is a sequence of abstract rules, separated by “|”s, and containing at least one rule. Each abstract rule follows this format:

$\langle rhs \rangle = \langle IR-tree-template \rangle$

A *rhs* is any legal *rhs* as described in Section 3.2. Note that only abstract non-terminals and the terminal lexemes are legal grammar symbols in an abstract *rhs*.

An *IR-tree-template* describes how to represent as a tree the syntax defined by an abstract rule. An IR template must provide three pieces of internal tree representation information. First, it must specify whether the abstract rule is represented in the tree by a node, by an annotation, or not at all. Second, if the rule is represented by a node, the template can specify which of the symbols in the *rhs* of the rule are to have their sub-tree representations be children of this node. Finally, the template may specify the order in which these children should be represented. Each IR tree template should match one of

1. “IMPLICIT”
2. “ANNOTATE” $\langle child-spec \rangle$
3. “ANNOTATE”
4. $\langle identifier \rangle (\langle child-spec-list \rangle)$
5. $\langle identifier \rangle$
6. “TREE”

or may be left out, along with the preceding “=”.

If no template is specified, a default is assigned. Recall that some terminal lexemes are specified as preserved in the lexical section. All non-terminals are by definition preserved. The set of preserved grammar symbols is information required by many of the tree templates.

A *child-spec-list* is simply a sequence of *child-specs* separated by “,”s. This sequence may be empty. A *child-spec* is a grammar symbol, optionally followed by an integer in angle brackets (“<”, “>”). Each *child-spec* must refer to a grammar symbol in the *rhs* preceding it. A *child-spec-list* must refer to all of the non-terminals in the *rhs* and each grammar symbol in the *rhs* may be referred to at most once. The *child-spec-list* may order the grammar symbols arbitrarily, however. The bracketed integer should be included after symbols that occur multiple times in the *rhs*; a 1 refers to the first occurrence, and so forth. A warning is issued if an omitted lexeme is part of the *child-spec-list*, or if a preserved non-constant lexeme is part of the *rhs* but is not in the *child-spec-list*.

Each rule with tree template 1 must have exactly one grammar symbol in its *rhs*, and must not be a sequence rule. The rule is not represented explicitly in the tree. Its place is held by the node representing the unique symbol in its *rhs*.

A rule that uses template 2 is represented as an annotation in the node representing the symbol referred to by the *child-spec* argument. The *child-specs* in these templates must be single element *child-spec-lists*; warnings will be issued accordingly. A rule using template 3 must have exactly one symbol in its *rhs* that is preserved, or have only one symbol in its *rhs* at all. This template is equivalent to template 2 with that unique symbol as the argument.

Tree template 4 is the most general. It specifies that the rule containing the template is represented by a tree node, that the *identifier* is the name for the rule and node, and that the children of the node are the sub-trees representing the grammar symbols in the *child-spec-list*, in the order specified. Template 5 is equivalent to the general template with the *identifier* as the node name argument and the preserved symbols of the preceding *rhs* in the order they appear in the *rhs* as the child list argument. The name of a node as specified by either of these templates must be unique, and must not be the name of a lexeme or an abstract non-terminal. An abstract non-terminal definition with only one rule may use template 6. This template is equivalent to template 5 with the name of the non-terminal as its node name argument.

If no tree template is specified for a rule, a default template is assigned as follows: If the rule's *rhs* is exactly one symbol, the default is template 1. If the rule's *rhs* contains more than one preserved symbol, the default is template 5. The name used for the template is “*NT-n*”, where the rule is the *n*th rule for non-terminal *NT*. Otherwise, the default is template 3. A warning is issued for the latter two kinds of default templates.

Each sequence rule must be represented by an actual tree node. Exactly one symbol in each sequence rule must have its corresponding sub-tree specified as a child of the rule's node. This is so that the sequence as a whole can be represented as a single node that has as its children exactly one node for each element of the sequence. Note that a sequence rule may have no more than one non-terminal in its *rhs* as a consequence. Further, no sequence delimiter may ever be explicitly represented in an IR tree.

The rule $\hat{A} \rightarrow [\hat{\alpha}]$ is shorthand for the two rules $\hat{A} \rightarrow \hat{\alpha}$ and $\hat{A} \rightarrow \epsilon$. The IR tree template specified for $\hat{A} \rightarrow [\hat{\alpha}]$ is assigned to rule $\hat{A} \rightarrow \hat{\alpha}$. Rule $\hat{A} \rightarrow \epsilon$ is given a special tree template that represents it by a distinct node for the rule. This node has the name “EMPTY”, and naturally has no children.

Ladle defines the cyclic rules \hat{P}_{cyclic} as the set of all abstract rules whose *rhs* is the rule's *lhs* plus one or more lexemes and whose IR template is ANNOTATE or IMPLICIT, with the only first rule for a given non-terminal included when there is more than one such rule. For most languages, there is at most one such rule for each abstract non-terminal.

3.4 The Concrete Section

The concrete section specifies an LALR parser for conversion of lexical streams to concrete derivations that can in turn be *Contracted* to form abstract derivations. This parser is specified by an extended LALR(1) context-free grammar \mathcal{G} that must be an expansion of the abstract grammar $\hat{\mathcal{G}}$. The definition of expansion places such strong constraints on the two grammars that the concrete grammar is usually very similar to the abstract grammar. Therefore, Ladle requires each abstract

non-terminal to have the same name as the concrete non-terminal to which it corresponds, that is, $\mathcal{A}_{base} = \hat{N}$ and $\psi_0 \cap (\mathcal{A}_{base} \times \hat{N}) = 1_{\hat{N}}$. Thus, \mathcal{G} must be an expansion of the abstract grammar $\hat{\mathcal{G}}$ relative to the set of cyclic abstract rules \hat{P}_{cyclic} specified in the abstract section and the mapping $\psi_0 = 1_{(\hat{\Sigma} \cup \hat{N})}$. Further, since the grammars are usually so similar, the concrete section of the language specification need not specify grammar rules for any non-terminal in \mathcal{A}_{base} whose rules in \mathcal{G} correspond exactly to its rules in $\hat{\mathcal{G}}$. All non-terminals in \mathcal{A}_{base} are assumed by the Ladle processor to have concrete rules that correspond to the non-terminal's abstract rules. The concrete section must contain a set of rules for each non-terminal in $N \setminus \mathcal{A}_{base}$. The concrete section should also contain an explicit set of rules for non-terminals in \mathcal{A}_{base} whose concrete rules do not exactly correspond to their abstract rules. Rule specifications of this latter kind override the default set of rules constructed from the abstract grammar. The concrete section consists of a set of definitions, each of which specifies the rules for one concrete non-terminal.

A *concrete-definition* has the form:

$\langle identifier \rangle \text{ ``=" } \langle concrete-rule-seq \rangle \text{ ``;"}$

The *identifier* names the non-terminal, and must not be the name of a lexeme. A non-terminal may be defined at most once in the concrete section. The *concrete-rule-seq* is a sequence of concrete rules, separated by “|”s, and containing at least one rule. Each such rule must be a legal *rhs* as described in Section 3.2. The set of legal grammar symbols for a concrete *rhs* consists of the terminal lexemes and any non-terminal defined in either the abstract or concrete sections.

4 Output

The Ladle processor outputs the data needed to convert between text and syntax trees, and to manipulate syntax trees directly, for a specified language. This section describes that data at a fairly high level. The details of the representation can be found in the implementation. The various symbols, rules, states, and so forth are all represented in the output by integers. The data should only be accessed indirectly, using the interface specified in Section 5.

The output includes the name of the language.

4.1 Lexical Data

A lexical automaton is output. This automaton is an encoding of all the lexemes' extended regular expressions. It can be used to extract the lexemes from a text stream. This description is intentionally left vague, as it is very implementation dependent.

For each lexeme, its text, whether it is ignored, screened, or parsed, and its size are output. The text of a constant lexeme is its expression string. For a case insensitive string, the cases of the characters in the text are those given in the string's original specification. The text of a non-constant lexeme is its name. The size of a constant lexeme is the length of its string. Non-constant lexemes are assigned a size of 0.

4.2 Abstract Syntax Tree Data

For each abstract non-terminal $\hat{A} \in \hat{N}$, its name, size, and the set of abstract rules $lhs^{-1}(\{\hat{A}\})$ are output. The size is the number of characters in its name.

For each abstract rule \hat{p} , its name, arity, kind, *lhs*, *rhs* length, *rhs*, delimiter, and the IR permutation of its *rhs* are output. The *rhs* of a sequence rule is considered to be only the left operand of the sequence operator; the operator and the delimiter are not included. Special values are used for the delimiter of non-delimited sequence rules and non-sequence rules.

The name and arity of each abstract rule are both determined by the IR tree template associated with it. Some of the grammar symbols in the *rhs* of each abstract rule are part of the template associated with the rule. The arity of the rule is the number of such symbols in its *rhs*; All sequence rules are thus assigned an arity of one. The name of an IMPLICIT or ANNOTATE rule is “IMPLICIT” or “ANNOTATE”, respectively. The ϵ rules specified by the optional notation are named “EMPTY”. Every other abstract rule is named by the IR tree template associated with it.

The kind of each abstract rule is one of the following:

implicit
annotate
normal
star
plus
plural

All rules with IMPLICIT or ANNOTATE IR tree templates are **implicit** or **annotate** rules, respectively. The kind of a sequence rule with operator “*”, “+”, or “++” is **star**, **plus**, or **plural**, respectively. All other abstract rules are of kind **normal**. Sequence rules are always represented by explicit tree nodes, so an IMPLICIT or ANNOTATE sequence rule is impossible.

The IR permutation of the *rhs* of each abstract rule relates the order of the symbols in the rule’s *rhs* to the order of the child specifications in the IR tree template associated with the rule. For each child specification in the rule’s template the rule’s IR permutation gives the *rhs* index of the specified grammar symbol.

The *Expand Top*, and *Bottom* mappings are output.

4.3 Parsing and Unparsing Data

Theoretically, a lexical stream can be converted to a syntax tree by parsing and contracting. A concrete derivation is constructed from a lexical stream by an LALR parser. Applying *Contract* to the concrete derivation yields an abstract derivation, which is then represented by a syntax tree. Unfortunately, neither LALR parsing nor the *Contract* algorithm given in Section 2.4 are defined on extended grammars. To handle these problems, a non-extended LALR parse grammar \mathcal{P} such that

$Lang(\mathcal{P}) = Lang(\mathcal{G})$ is constructed, and the *Contract* algorithm is adjusted to apply to derivations of \mathcal{P} rather than \mathcal{G} .

The context-free parse grammar $\mathcal{P} = \langle N^*, \Sigma, P^*, S \rangle$ is a normalization of the concrete grammar \mathcal{G} . P^* is created from P by normalizing all sequence rules. N^* is N plus whatever extra non-terminals are required for the normalized rules. Σ and S are the same as in \mathcal{G} . For each normal transformation, let $A \in N$, $\alpha \in (\Sigma \cup N^*)^*$ such that $|\alpha| \neq 0$, and $d \in \Sigma \cup \{\epsilon\}$ be given. For each sequence rule, let a distinct $X \notin N$ be given. The transformations are

$A \rightarrow \alpha * d$	becomes	$A \rightarrow \epsilon$	(abstract)
		$A \rightarrow X$	
		$X \rightarrow \alpha$	(abstract)
		$X \rightarrow X d \alpha$	(enqueue)
$A \rightarrow \alpha + d$	becomes	$X \rightarrow X d \overline{X}$	(append)
		$A \rightarrow X$	
		$X \rightarrow \alpha$	(abstract)
		$X \rightarrow X d \alpha$	(enqueue)
$A \rightarrow \alpha ++ d$	becomes	$X \rightarrow X d \overline{X}$	(append)
		$A \rightarrow X$	
		$X \rightarrow \alpha d \alpha$	(plural)
		$X \rightarrow X d \alpha$	(plural)
		$X \rightarrow X d \overline{X}$	(plural).

The symbol X defined by each application of a transformation is an element of N^* . \overline{X} is *not* a distinct symbol from X , but is a specification needed for LALR automaton generation, since the parse grammar may be ambiguous. These transformations are the same as those in Appendix C except for the addition of the rules containing \overline{X} , which are explained in the next section. The parenthesized keywords after some of the transformation rules are used by the adjusted *Contract* algorithm described shortly.

Using the normalization given in Appendix B on $\hat{\mathcal{G}}$ and \mathcal{G} yields the context-free grammars $\hat{\mathcal{G}}'$ and \mathcal{G}' such that \mathcal{G}' is LALR(k) and an expansion of $\hat{\mathcal{G}}'$. Unfortunately, applying *Contract* to a derivation of \mathcal{G}' may result in a derivation of $\hat{\mathcal{G}}'$ that is different from the appropriate derivation of $\hat{\mathcal{G}}$. For example, a $\hat{\mathcal{G}}$ derivation tree represents each sequence with a single rule node whose children are the sequence elements, while a $\hat{\mathcal{G}}'$ derivation tree represents the same sequence with a left-recursive binary tree whose leaves are the sequence elements. However, the sequence of leaves in any binary tree can be constructed bottom-up as follows:

- For each left leaf node, create the sequence containing the leaf: **abstract**.
- For each internal node whose right child is a leaf node, add the right child leaf to the sequence representing the left child sub-tree's leaves: **enqueue**.
- For each internal node whose right child is another internal node, append the sequences representing the left and right child sub-trees' leaves: **append**.

Using this method, the *Contract* algorithm can be extended to convert binary trees to sequences.

Let the *action'* and *data'* mappings for $\hat{\mathcal{G}}'$ and \mathcal{G}' be given. Note that the rules of \mathcal{P} are a superset

of the rules of \mathcal{G}' .

Define $action^* : P^* \rightarrow \{\text{none, mark, abstract, disambiguate, enqueue, append, plural}\}$ by

$$\forall p \in P^*, \quad action^*(p) = \begin{cases} \text{abstract} & \text{if } p \text{ was defined as abstract} \\ \text{enqueue} & \text{if } p \text{ was defined as enqueue} \\ \text{append} & \text{if } p \text{ was defined as append} \\ \text{plural} & \text{if } p \text{ was defined as plural} \\ \text{abstract} & \text{if } data(p) \in \Pi \text{ and } Contract(data(p)) \in \hat{P} \\ \text{none} & \text{if } data(p) \in \Pi \text{ and } Contract(data(p)) \notin \hat{P} \\ action'(p) & \text{otherwise} \end{cases}$$

where “defined as” refers to the keyword to the right of the rule as it was defined in the normal transformations. For “++” sequences there is no abstract operation, and the plural operation performs either enqueue or append as appropriate. Note that the abstract operation sometimes takes the place of a mark operation. This facilitates splitting a concrete derivation tree into subtrees which represent derivations in Π . Define the $data^*$ mapping for \mathcal{P} from $data'$ by substituting the original abstract sequence rule for each abstract rule produced by normalizing, wherever such rules appear.

Under some circumstances, more optimal versions of the concrete to parse grammar transformations are applicable. The transformations

$$\begin{array}{lll} A \rightarrow \alpha* & \text{becomes} & \begin{array}{ll} A \rightarrow X & \\ X \rightarrow \epsilon & (\text{abstract}) \\ X \rightarrow X\alpha & (\text{enqueue}) \\ X \rightarrow X\overline{X} & (\text{append}) \end{array} \\ \\ A \rightarrow \alpha++d & \text{become} & \begin{array}{ll} A \rightarrow X & \\ X \rightarrow \alpha & (\text{abstract}) \\ X \rightarrow Xd\alpha & (\text{plural}) \\ X \rightarrow Xd\overline{X} & (\text{plural}) \end{array} \\ A \rightarrow \alpha & & \end{array}$$

are used in place of the standard transformations whenever possible. If any of the transformations results in $A \rightarrow X$ being the only rule for A , that rule may be omitted, and A used in place of X . This optimization is not applicable to delimited “*” sequences. The $action^*$ and $data^*$ mappings must be adjusted for these optimizations.

A special LALR automaton ACTION table is constructed for \mathcal{P} . This table differs from conventional ones in that the ACTION mapping is defined not only on terminals, but on non-terminals as well. So defined, this mapping subsumes the GOTO mapping entirely. With this alteration, the LALR ACTION table can be used to recognize not only strings of the language, but also sentential forms. Note that the inclusion of the \overline{X} rules permits the recognition of a non-terminal representing a sub-sequence at any point within a sequence. Without those rules, such non-terminals could only be recognized at the beginnings of sequences.

The presence of the \overline{X} rules in \mathcal{P} make that grammar ambiguous. This ambiguity is resolved in the construction of the ACTION table by treating \overline{X} non-terminals somewhat like terminals. \overline{X} indicates *only* that the non-terminal X is acceptable in this position. It does *not* indicate that an

α such that $X \xRightarrow{*} \alpha$ is also acceptable. Stated in the jargon of LALR parsing, in this context X can be shifted, but not obtained via a reduction. With this restriction, an LALR ACTION table can be constructed for \mathcal{P} without changing the intended semantics of the \overline{X} rules.

The LALR ACTION table is output. It can be used to parse a concrete phrasal form that derives from any given non-terminal. All that is necessary is the start state for the non-terminal and a terminal symbol in the non-terminal's follow set. Both of these are output for each concrete non-terminal in \mathcal{A}_{top} .

The sets Σ , \hat{N} , \mathcal{A} , \mathcal{A}_{chain} , N_{ambig} , and N are output.

The mappings $\psi \cap (\mathcal{A}_{chain} \times \hat{N})$, $action^*$, and $data^*$ are output.

For each parse rule $p \in P^*$, $lhs(p)$ and $|rhs(p)|$ are output.

4.4 Data Representation

The integers that represent the symbols and rules in the abstract and parse grammars are carefully chosen to simplify Ladle table access. Since $\psi \cap (\mathcal{A}_{top} \times \hat{N})$ provides such a strong relationship between the abstract non-terminals in \hat{N} and the concrete/parse non-terminals in \mathcal{A}_{top} , the corresponding abstract and parse non-terminals in those sets are numbered identically. The concrete non-terminals in \mathcal{A} are ordered so as to map \prec onto $<$. The abstract rules for each abstract non-terminal are ordered with all rules in \hat{P}_{cyclic} before all other rules. The abstract rules as a whole are ordered by the order on their lhs 's. The concrete states Π_{sub} are numbered so that $\forall \pi \in \Pi$, if $Contract(\pi) \in \hat{P}$, then π is numbered identically to $Contract(\pi)$, otherwise π is numbered 0. The symbols and rules are indexed as follows:

The ignored lexemes are not indexed.

The non-constant screened lexemes are 1 through $K - 1$.

The constant screened lexemes are K through $L - 1$.

The constant preserved or omitted lexemes are L through $V - 1$.

The non-constant preserved or omitted lexemes are V through $S - 1$.

The non-terminals in \hat{N} (or in \mathcal{A}_{top}) are S through $R - 1$, with \hat{S} (or $Top(S)$) first.

The non-terminals in $\mathcal{A} \setminus \mathcal{A}_{top}$ are R through $A - 1$.

The non-terminals in $\mathcal{A}_{chain} \setminus \mathcal{A}$ are A through $C - 1$.

The non-terminals in $N_{ambig} \setminus \mathcal{A}_{chain}$ are C through m .

The non-terminals in $N^* \setminus N_{ambig}$ are $m + 1$ through n .

The abstract grammar rules \hat{P} are R through $U - 1$.

The parse grammar rules P^* are 1 through r .

Figure 3 illustrates these numeric assignments. (U may or may not actually be less than C , m , or n , but all of the other variables are ordered correctly.) Note that all lexemes, abstract non-terminals,

1	K	L	V	S
screened lexemes		abstract lexemes		
var lexemes	constant lexemes	var lexemes		

	S	R	A	U	C	m	n
	abstract NTs (N)		abstract rules				
	\mathcal{A}						
	abstracted parse NTs (\mathcal{A}_{chain})					N_{ambig}	
	parse NTs (N^\bullet)						

Figure 3: Integer assignment for the grammars

and abstract rules are assigned distinct values. They are collectively called operators. Each node of a syntax tree IR has an operator that describes what the node represents.

$K, L, V, S, R, A, C, U, m, n$ and r are all included in Ladle's output. Note that the abstract start symbol \hat{S} and the concrete start symbol representative $Top(S)$ are both numbered S .

For each abstract non-terminal, define $first(\hat{X})$ as the index of the first rule with \hat{X} as its *lhs*. Define $first(R)=U$, where R and U are the indices defined earlier. The set of rules for each abstract non-terminal \hat{X} is thus represented by the set of integers $[first(\hat{X}), first(\hat{X} + 1)[$. *first* is the representation of these sets used in the output.

Expand is represented by $(lhs \circ Expand) \cap (\hat{P} \times P)$, $(rhs \circ Expand) \cap (\hat{P} \times P)$, and $(Contract \circ Coerce')$. For each abstract rule $\hat{p} \in \hat{P}$, $rhs(Expand(\hat{p}))$ may be parsed into $lhs(Expand(\hat{p}))$ to yield $Expand(\hat{p})$. Using $(Contract \circ Coerce')$, this technique may be generalized from single abstract rules to any non-null abstract derivation.

For each abstract rule $\hat{p} \in \hat{P}$, $\alpha = lhs(Expand(\hat{p}))$ and $A = rhs(Expand(\hat{p}))$ are output. Since $\psi(A) = lhs(\hat{p})$ and $\psi(\alpha) = rhs(\hat{p})$, A represents both $lhs(\hat{p})$ and $(lhs \circ Expand)(\hat{p})$, and α represents both $rhs(\hat{p})$ and $(rhs \circ Expand)(\hat{p})$.

$(Contract \circ Coerce')$ is completely represented by the order of the concrete non-terminals and the abstract rules, since \prec is represented by $<$ and each rule $\hat{p} \in \hat{P}_{cyclic}$ is represented by $first(lhs(\hat{p}))$.

$\psi \cap ((\Sigma \cup \mathcal{A}_{top}) \times (\hat{\Sigma} \cup \hat{N}))$ is represented by 1_N , so $\psi \cap ((\Sigma \cup \mathcal{A}_{chain}) \times (\hat{\Sigma} \cup \hat{N}))$ is represented in the output by $\psi \cap ((\mathcal{A}_{chain} \setminus \mathcal{A}_{top}) \times \hat{N})$ only. The representation of $\psi \cap (\mathcal{A}_{top} \times \hat{N})$ represents *Top* as well.

The *action*^{*} and *data*^{*} mappings output are modified to ignore implicit abstract rules and null abstract derivations whenever possible, since the IR tree doesn't include them.

Bottom and $(lhs \circ Expand) \cap (\hat{P} \times P)$ are represented by a single vector that includes $1_{\hat{\Sigma}}$. This vector maps each operator $\hat{X} \in (\hat{\Sigma} \cup \hat{N} \cup \hat{P})$ onto a parse symbol X that can be derived, using

only trivial derivations, from the *lhs* of the expansion of the abstract derivation represented by any syntax tree whose root operator is \hat{X} . For all $\hat{X} \in (\hat{\Sigma} \cup \hat{N} \cup \hat{P})$, $lhs(Expand(\hat{X})) \xrightarrow[\Pi_{trivial}]{*} X$, since for all $\hat{X} \in (\hat{\Sigma} \cup \hat{N})$, \hat{X} is isomorphic to $\emptyset_{\hat{X}}$.

The names of lexemes, abstract non-terminals, and abstract rules are all combined in a single vector. The sizes of lexemes and abstract non-terminals are gathered together in the same vector as the arities of abstract rules.

5 Client Interface

The Ladle client interface provides a simple means of accessing the output tables. In the interface, the names *symbol*, *non-terminal*, *NT*, and *rule* refer to the abstract grammar, unless otherwise noted. Grammar symbols, rules, states, and so forth are represented by integers. No distinction is made between the integer and what it represents.

5.1 Language Forms

(load-language *language-name*)

Load and return the Ladle tables for the language named *language-name*. There may be optional extra parameters to this function that specify system dependent load arguments.

(with-language *language*
body)

Make *language* the current language while executing *body*. Many of the other forms and functions use the current language.

(current-language)

Return the current language, or false if there is none.

(language-name)

Return the current language's name.

(language-top-operator)

Return the abstract and parse grammar start symbol \hat{S} .

(do-operators (*variable return-form*)
body)

For each operator in the language, bind *variable* to the operator and execute *body*. The *return* function may be used in the body as in all loops. Return the result of *return-form*, or true if no *return-form* is given.

(do-lexemes (*variable return-form*)
body)

For each lexeme in the language, bind *variable* to the lexeme's operator and execute *body*. The *return* function may be used in the body as in all loops. Return the result of *return-form*, or true if no *return-form* is given.

(do-NTs (*variable return-form*)
body)

For each abstract non-terminal, bind *variable* to the non-terminal's operator and execute *body*. The *return* function may be used in the body as in all loops. Return the result of *return-form*, or true if no *return-form* is given.

5.2 Lexer Data Forms

initial-lex-state

The initial lex state and first argument to *lex-action*.

(lex-action *lex-state character*)

Return the lex action for the *character* when in the *lex-state*: False if an ignored lexeme has been recognized, a lexeme if one that is not ignored has been recognized, the new state otherwise. The lexeme may be *lexical-error-operator* or *eos-lexeme-operator*. Note that a lexeme only specifies the lexical class; the characters recognized are not preserved here.

5.3 Parser Data Forms

(initial-state-for-parse-NT *parse-NT*)

The initial parse state and first argument to *parse-action* when parsing a phrasal form derived from the *parse-NT*. It is an error for *parse-NT* $\notin \mathcal{A}_{top}$.

(parse-action *parse-state parse-symbol*)

Return the LALR parse ACTION for the *parse-symbol* when in the *parse-state*: a new state for shift, a parse rule for reduce, or false for error.

(follow-lexeme-for-parse-NT *parse-NT*)

Return any lexeme in the follow set of the *parse-NT*. It is an error for *parse-NT* $\notin \mathcal{A}_{top}$.

(parse-rule-lhs *parse-rule*)

Return the parse non-terminal *lhs(parse-rule)*.

(parse-rule-length *parse-rule*)

Return the length of the *parse-rule*.

(parse-rule-action *parse-rule*)

Return *action*(parse-rule)*.

(parse-rule-data *parse-rule*)

Return $data^*(parse-rule)$.

(disambiguate-parse-action *parse-rule parse-rule-rhs-state-sequence*)

Return the disambiguated action for the *parse-rule*: a *Contract* state for **mark** or an abstract rule for **abstract**. *Parse-rule-rhs-state-sequence* contains the *Contract* states corresponding to each symbol in the *parse-rule*'s *rhs*.

(parse-symbol-operator *parse-symbol*)

Return $\psi(parse-symbol)$. *Parse-symbol* must be an element of $(\Sigma \cup \mathcal{A}_{chain})$.

(operator-parse-symbol *operator*)

Return $lhs(Expand(operator))$, where for all $\hat{X} \in (\hat{\Sigma} \cup \hat{N})$, \hat{X} is isomorphic to $\emptyset_{\hat{X}}$.

(list-parse-NT-to-NT-rule-cycle *from-parse-NT to-parse-NT*)

Return the derivation $Contract(Coerce'(from-parse-NT, to-parse-NT))$ as a list of abstract rules. This list may be destructively modified. It is an error if *from-parse-NT* or *to-parse-NT* is not in \mathcal{A} , or if $\psi(from-parse-NT) \neq \psi(to-parse-NT)$.

5.4 Generic Operator Forms

syntactic-error-operator

lexical-error-operator

eos-lexeme-operator

Each syntactic or lexical error is represented by the *syntactic-error-operator* or the *lexical-error-operator*, respectively. *Eos-lexeme-operator* represents the end of the lexical stream. It is returned by *lex-action* and expected by *lalr-action*. These operators are collectively called the special operators.

(find-named-operator *operator-name*)

Return the integer assigned to the operator named *operator-name*, or false if there is no such operator.

(operator-text *operator*)

Return the text for *operator*. It is an error for *operator* to be special. If *operator* is a constant lexeme, the lexeme itself is returned. Otherwise, the name of the lexeme, non-terminal, or rule is returned.

(operator-is-special? *operator*)

Return true if *operator* is special, false otherwise.

(operator-is-constant? *operator*)

Return false if *operator* is a non-constant lexeme, true otherwise.

(operator-is-screened? *operator*)

Return true if *operator* is a screened lexeme, false otherwise.

(operator-is-lexeme? *operator*)

Return true if *operator* is a lexeme or is special, false otherwise.

(operator-is-symbol? *operator*)

Return false if *operator* is a rule, true otherwise.

(operator-is-variable-arity? *operator*)

If *operator* is a delimited sequence, return the sequence's delimiter. If *operator* is the syntactic-error-operator or an undelimited sequence, return syntactic-error-operator. If the arity of *operator* is fixed (possibly zero), return nil.

5.5 Symbol Forms

(symbol-size *symbol-operator*)

For a non-terminal, return the number of characters in its name. For a constant lexeme, return the number of characters in its representation. For non-constant lexemes, return 0. It is an error if *symbol-operator* is a special operator.

(do-NT-rules (*variable NT-operator return-form*)

body)

For each abstract rule \hat{p} for which $lhs(\hat{p}) = NT\text{-}operator$, bind *variable* to the rule's operator and execute *body*. The **return** function may be used in the body as in all loops. Return the result of *return-form*, or true if no *return-form* is given.

(NT-bottom *NT-operator*)

Return *Bottom(NT-operator)*.

5.6 Rule Forms

(rule-kind *rule-operator*)

Return the kind of abstract rule *rule-operator*.

(rule-lhs *rule-operator*)

Return $lhs(Expand(\hat{p}))$, where $\psi(lhs(Expand(\hat{p}))) = lhs(\hat{p})$ and abstract rule \hat{p} is represented by *rule-operator*.

(rule-arity *rule-operator*)

Return the arity of abstract rule *rule-operator*.

(do-rule-rhs-operators-and-child-indices ((*symbol-var child-var*) *rule-operator return-form*)

body)

Let $\alpha = rhs(Expand(\hat{p}))$, where $\psi(\alpha) = rhs(\hat{p})$ and \hat{p} is represented by *rule-operator*. For each symbol in α , execute *body* with *symbol-var* bound to the symbol's operator and *child-var* bound to the index of the

corresponding child specification in the rule's IR tree template, or -1 if there is none. The `return` function may be used in the body as in all loops. Return the result of *return-form*, or true if no *return-form* is given.

A Notation and Conventions

A.1 Notation

Let S and T be sets.

Define $S \setminus T$ as the set difference of S and T .

Define $1_S : S \rightarrow S$ as the identity mapping on S .

Let S^* denote the application of the Kleene star operator to S .

Define $\text{Subsets}(S) = \{S' \mid S' \subseteq S\}$.

Use the notation $\exists! x \dots$ to denote “there exists a unique x such that \dots ”.

Let $f : D \rightarrow C$ be a mapping.

If f is invertible, define $f^{-1} : C \rightarrow D$ as the f inverse mapping.

Define the f image mapping $f : \text{Subsets}(D) \rightarrow \text{Subsets}(C)$ by

$$\forall S \subseteq D, \quad f(S) = \{f(d) \mid d \in S\}.$$

Define the f preimage mapping $f^{-1} : \text{Subsets}(C) \rightarrow \text{Subsets}(D)$ by

$$\forall S \subseteq C, \quad f^{-1}(S) = \{d \mid f(d) \in S\}.$$

Let the subsets $D' \subseteq D$ and $C' \subseteq C$ be given such that $f(D') \subseteq C'$.

Define the restriction mapping $f \cap (D' \times C')$ by

$$\forall d \in D', \quad f \cap (D' \times C')(d) = f(d).$$

The empty string is represented by ϵ .

Let $f : D \rightarrow C$ be a mapping, where D and C are sets of symbol strings and $\epsilon \notin D$.

Extend f homomorphically to the mapping $f : D^* \rightarrow C^*$ by

$$f(\alpha\beta) = f(\alpha)f(\beta) \quad \text{and} \quad f(\epsilon) = \epsilon.$$

A *context-free grammar* \mathcal{G} is a tuple $\langle N, \Sigma, P, S \rangle$, where N is the set of non-terminal symbols, Σ is the set of terminal symbols, P is a set of rewrite rules, and $S \in N$ is the initial symbol.

A *rewrite rule* in P is written $\langle A \rightarrow \alpha \rangle$, where $A \in N$ and $\alpha \in (\Sigma \cup N)^*$.

A is the *left-hand-side* or *lhs*, and α is the *right-hand-side* or *rhs*.

A *chain rule* is any rule of the form $\langle A \rightarrow B \rangle$, where $B \in N$. An *empty rule* is any rule of the form $\langle A \rightarrow \epsilon \rangle$.

For any $M \subseteq N$, define $P_{-M} \subseteq P$ by $P_{-M} = \{\langle A \rightarrow \alpha \rangle \in P \mid A \notin M\}$.

$\alpha \xrightarrow[P']{\pi} \beta$ denotes a *derivation* that rewrites $\alpha \in (\Sigma \cup N)^*$ as $\beta \in (\Sigma \cup N)^*$ by applying a sequence π of rewrite rules in $P' \subseteq P$.

In general, a derivation may be of any length, including zero.

For any grammar symbol $A \in (\Sigma \cup N)$, define \emptyset_A as the zero length derivation for which $A \xRightarrow{\emptyset_A} A$.

A zero length derivation is also called a *null* derivation.

$\alpha \Rightarrow \beta$ denotes a derivation consisting of exactly one rule.

$\alpha \xRightarrow{*} \beta$ denotes a derivation without naming it.

P' may actually be a set of derivations rather than of rules. If P' is not specified, P is assumed.

When $\alpha \xRightarrow{*} \beta$ and $\alpha \in N$, β is called a *phrasal form*. When $\alpha = S$, β is called a *sentential form*.

Let $\langle \alpha \xRightarrow{\pi_0} \beta \rangle$ and $\langle \delta \xRightarrow{\pi_1} \gamma \rangle$ be derivations such that $\beta = \xi\delta\zeta$, for some $\alpha, \beta, \delta, \gamma, \xi, \zeta \in (\Sigma \cup N)^*$.

Then $\langle \alpha \xRightarrow{\pi_0\pi_1} \xi\gamma\zeta \rangle$ is a *concatenation* of π_0 and π_1 .

Note that there may be many ways to concatenate two derivations.

Let Π_0 and Π_1 be sets of derivations.

Define $\Pi_0\Pi_1$ as the set of derivations that can be constructed by concatenating a derivation in Π_0 with a derivation in Π_1 .

Define Π_0^* as the set of derivations that can be constructed by concatenating zero or more derivations in Π_0 .

Define Π_0^+ as the set of derivations that can be constructed by concatenating one or more derivations in Π_0 .

The mappings $lhs : P^* \rightarrow N$ and $rhs : P^* \rightarrow (\Sigma \cup N)^*$ are defined by

$$\forall \pi \in P^*, \text{ where } A \xRightarrow{\pi} \alpha, A \in N, \text{ and } \alpha \in (\Sigma \cup N)^*, \quad \begin{array}{l} lhs(\pi) = A \\ rhs(\pi) = \alpha. \end{array}$$

Let $\pi \in P^*$ be a derivation and $P' \subseteq P$ be a set of rules.

Define the derivation $\pi \setminus P'$ as the derivation π with all of the rules in P' removed.

Care must be exercised with this operation, as $\pi \setminus P'$ is not always actually a derivation.

For any grammar $\mathcal{G} = \langle N, \Sigma, P, S \rangle$, $Lang(\mathcal{G})$ is the set of terminal strings that can be derived from S using the rules in P .

A.2 Conventions

\mathcal{G} is a context-free grammar.

A, B, C , and X are non-terminals.

p and q are rewrite rules.

$\alpha, \beta, \gamma, \delta, \xi$, and ζ are strings of terminals and non-terminals.

π is a derivation.

\perp is a symbol such that $\perp \notin (\Sigma \cup N)$.

All symbols relating to abstract grammars have hats on them; e.g. \hat{A} , \hat{p} , $\hat{\alpha}$, $\hat{\pi}$, and so forth. Conversely, no symbol that does *not* relate to abstract grammars has a hat on it.

B Ladle in Ladle

LANGUAGE ladle

/* A Ladle description of the Ladle input syntax. */

LEXICAL

whitespace = { \t\n\L } => IGNORE;

comment = "/*" ~ "*/" => SCREEN;

string = "\"" - "\"" ;

case_insensitive_string = "'" - "'" ;

char_set = "[" - "]" ;

index = "<" {0-9}+ ">" ;

identifier = {a-zA-Z}({_0-9a-zA-Z})*;

key_abstract = 'ABSTRACT' IN identifier;
key_annotate = 'ANNOTATE' IN identifier;
key_concrete = 'CONCRETE' IN identifier;
key_ignore = 'IGNORE' IN identifier;
key_implicit = 'IMPLICIT' IN identifier;
key_in = 'IN' IN identifier;
key_language = 'LANGUAGE' IN identifier;
key_lexical = 'LEXICAL' IN identifier;
key_omit = 'OMIT' IN identifier;
key_preserve = 'PRESERVE' IN identifier;
key_screen = 'SCREEN' IN identifier;
key_tree = 'TREE' IN identifier;

ABSTRACT

ladle_specification = 'LANGUAGE' identifier
 'LEXICAL' lexical_definition_seq
 'ABSTRACT' abstract_definition_seq
 opt_concrete_section => TREE;

opt_concrete_section = ['CONCRETE' concrete_definition_seq] ;

```

lexical_definition_seq = lexical_definition* => TREE;

lexical_definition = identifier "="
lexical_expr lexical_disposition ";" => TREE;

lexical_expr = lexical_expr ++ "|"      => or
               | lexical_expr ++        => concatenate
               | lexical_expr "-" string => match
               | string "~" string      => balanced_match
               | any_string 'IN' identifier => lex_in
               | lexical_expr "*"        => lex_star
               | lexical_expr "+"        => lex_plus
               | "[" lexical_expr "]"    => lex_optional
               | "(" lexical_expr ")"    => ANNOTATE
               | string
               | case_insensitive_string
               | char_set
               ;

any_string = string
            | case_insensitive_string
            ;

lexical_disposition =                => l_default_disposition
                    | ">" 'IGNORE'    => l_ignore
                    | ">" 'SCREEN'    => l_screen
                    | ">" 'OMIT'      => l_omit
                    | ">" 'PRESERVE' => l_preserve
                    ;

rhs = grammar_symbol_seq "*" opt_grammar_symbol => star
      | grammar_symbol_seq "+" opt_grammar_symbol => plus
      | grammar_symbol_seq "++" opt_grammar_symbol => plural
      | "[" grammar_symbol_seq "]"                => optional
      | grammar_symbol_seq
      ;

grammar_symbol_seq = grammar_symbol* => TREE;

opt_grammar_symbol = [ grammar_symbol ]
                    ;

grammar_symbol = string
                | case_insensitive_string
                | identifier
                ;

```

```

abstract_definition_seq = abstract_definition+ => TREE;

abstract_definition = identifier "=" abstract_rule_seq ";" => TREE;

abstract_rule_seq = abstract_rule + "|" => TREE;

abstract_rule = rhs tree_template => TREE;

tree_template =
    | ">" 'IMPLICIT'                => default_template
    | ">" 'ANNOTATE' abstract_child  => implicit_template
    | ">" 'ANNOTATE'                => annotate_template
    | ">" 'ANNOTATE'                => annotate_default_template
    | ">" identifier "(" abstract_child_seq ")" => general_template
    | ">" identifier                => named_template
    | ">" 'TREE'                    => simple_template
    ;

abstract_child_seq = abstract_child * "," => TREE;

abstract_child = grammar_symbol opt_index => TREE;

opt_index = [ index ] => TREE;

concrete_definition_seq = concrete_definition* => TREE;

concrete_definition = identifier "=" concrete_rule_seq ";" => TREE;

concrete_rule_seq = rhs + "|" => TREE;

CONCRETE

lexical_expr = lexical_term_seq ++ "|"
    | lexical_term_seq
    ;

lexical_term_seq = lexical_term++
    ;

lexical_term = lexical_factor "-" string
    | string "~" string
    | any_string 'IN' identifier
    | lexical_factor
    ;

```

```
lexical_factor = lexical_primary "*"
                | lexical_primary "+"
                | lexical_primary
                ;

lexical_primary = "(" lexical_expr ")"
                | "[" lexical_expr "]"
                | string
                | case_insensitive_string
                | char_set
                ;
```

C Normalizing Extended Context-Free Grammars

The theory of expansion as described in Section 2 does not include the special sequence operators, defined in Section 3.2. A context-free grammar containing these operators can be converted into an equivalent context-free grammar without them by normalizing each of its sequence rules. For each normal transformation, let $A \in N$, $\alpha \in (\Sigma \cup N)^*$ such that $|\alpha| \neq 0$, and $d \in \Sigma \cup \{\epsilon\}$ be given. For each sequence rule, let a distinct $X \notin \Sigma$ be given. The transformations are:

$$\begin{array}{lll}
 A \rightarrow \alpha * d & \text{becomes} & \begin{array}{l} A \rightarrow \epsilon \\ A \rightarrow X \\ X \rightarrow \alpha \\ X \rightarrow X d \alpha \end{array} \\
 \\
 A \rightarrow \alpha + d & \text{becomes} & \begin{array}{l} A \rightarrow X \\ X \rightarrow \alpha \\ X \rightarrow X d \alpha \end{array} \\
 \\
 A \rightarrow \alpha ++ d & \text{becomes} & \begin{array}{l} A \rightarrow X \\ X \rightarrow \alpha d \alpha \\ X \rightarrow X d \alpha \end{array}
 \end{array}$$

Each new concrete non-terminal X defined by transforming a rule of a concrete grammar is mapped by ψ_0 onto the new abstract non-terminal \hat{X} defined by transforming the similar rule in the corresponding abstract grammar.